Tekniker och verktyg som identifierar mjukvarusårbarheter

En skanning av forskning publicerad mellan 2014 – 2024

Christian Gustavsson, Christian Vestlund, Viktor Andersson, Lovisa Nyholm, Casper Jensen och Daniel Eidenskog

FOI-R--5692--SE

December 2024

FOI

Christian Gustavsson, Christian Vestlund, Viktor Andersson, Lovisa Nyholm, Casper Jensen och Daniel Eidenskog

Tekniker och verktyg som identifierar mjukvarusårbarheter

En skanning av forskning publicerad mellan 2014 – 2024

Titel	Tekniker och verktyg som identifierar mjukvarusårbarheter – En skanning av forskning publicerad mellan 2014 – 2024
Title	Techniques and tools that identify software vulnerabilities - A scan of published research between 2014 – 2024
Rapportnr/Report no	FOI-R--5692--SE
Månad/Month	December
Utgivningsår/Year	2024
Antal sidor/Pages	80
ISSN	1650-1942
Uppdragsgivare/Client	Försvarsmakten
Forskningsområde	Informationssäkerhet
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E38549
Godkänd av/Approved by	Emil Hjalmarson
Ansvarig avdelning	Cyberförsvar och ledningsteknik

Bild/Cover	Shutterstock

Sammanfattning

Att testa och verifiera säkerhetsegenskaper hos mjukvara är ett forskningsområde med hög aktivitet. Denna studie skannar av forskningsområdet för åren 2014–2024 efter tekniker och verktyg som kan upptäcka mjukvarusårbarheter. Sammantaget inventeras 237 verktyg i rapporten, varav de sju med högst bedömd mognadsgrad ligger till grund för en djupare analys och beskrivning.

Trots att det finns omfattande forskning inom området innehåller den få verktyg med en hög mognadsgrad. Av alla verktyg som tas upp i rapporten så är de flesta inriktade på att identifiera enstaka typer av sårbarheter, vilket gör deras användningsområde begränsat. Forskning på senare år försöker hitta ett bredare angreppssätt, genom att samla flera tekniker till hybridverktyg eller genom att introducera maskininlärningsmodeller tränade att identifiera brister. Det återstår dock forskning innan de kan betraktas som beprövade tekniker.

Nyckelord: mjukvara, sårbarhet, teknik, verktyg, cybersäkerhet

Summary

Testing and verifying the security features of software is an active area of research. This study scans the research area from 2014 to 2024 for techniques and tools that can detect software vulnerabilities. In total, the report inventories 237 tools, of which the 7 with the highest assessed maturity level are the basis for a deeper analysis and description.

Despite the extensive research in the field, it contains few tools with a high level of maturity. Of all the tools discussed in the report, most focus on identifying individual types of vulnerabilities, which limits their scope of use. Recent research has been trying to find a broader approach by combining several techniques into hybrid tools or by introducing machine learning models trained to identify flaws. However, further research is needed before these can be considered proven techniques.

Keywords: software, vulnerability, technique, tool, cyber security

Innehåll

1 Inledning

Cybersäkerhetsincidenter inträffar dagligen och får ibland stora konsekvenser för organisationer och användare. Många incidenter beror på sårbarheter som uppstår av felaktiga konfigurationer i mjukvara eller mänskliga fel i utveckling av mjukvara. Skadliga program som nyttjar mjukvarusårbarheter, såsom WannaCry [1] och Pegasus [2], är också en orsak till omfattande incidenter.

Kostnaden för cybersäkerhetsincidenter varierar stort och de kan bli väldigt dyra. Amerikanska CISA:s rapport som utvärderat uppskattningar från flera andra underlag rapporterar att mediankostnaden för en incident varierar mellan 600 000 och 20 miljoner kronor [3].[1] I en samhälls- eller försvarskontext är konsekvenserna inte enbart monetära utan kan även leda till andra typer av effekter såsom minskat förtroende för samhällskritiska funktioner, lägre vårdkvalitet eller sämre försvarsförmåga. Försvarsmakten skriver om denna typ av följder i sin strategiska inriktning [4]:

> *Effekterna av ett cyberangrepp kan få stora konsekvenser för samhällsviktiga funktioner, kritiska informationstekniksystem och påverkar Försvarsmaktens förmåga att verka i de övriga domänerna.*[2]

Att utveckla säker mjukvara är en utmaning som försvåras av en ständigt ökande komplexitet. Allt fler plattformar, programspråk och ramverk används inom utveckling och förväntas samspela och interagera med varandra. I en skrivelse författad av myndigheter från fjorton länder (bland annat amerikanska NSA, tyska BSI, brittiska NCSC-UK och holländska NCSC-NL) beskrivs statisk och dynamisk kodanalys som en *best practice* för att implementera säker mjukvara från grunden [6]. I en studie av Karlzén m.fl. identifierades flera orsaker till varför mjukvarusårbarheter uppstår, varav bristande verktygsstöd är en tydligt bidragande faktor [7]. Rapporten pekar också ut att verktyg för att identifiera mjukvarusårbarheter är en viktig del i utvecklingen av säker mjukvara samt att det finns flera forskningsfrågor som behöver undersökas inom området.

Denna rapport beskriver en litteraturstudie av de senaste tio årens forskning om verktyg som kan användas för att identifiera mjukvarusårbarheter. Fokuset på akademisk forskning innebär en begränsning av mängden verktyg som har uppnått praktisk användbarhet då den nivån normalt sett ligger bortom vad forskningen siktar på. Samtidigt vore det svårt att genomföra motsvarande undersökning av kommersiella verktyg eftersom det ofta saknas publikt tillgänglig information om detaljerna i hur verktygen fungerar.

1.1 Syfte och mål

Rapporten ingår i ett projekt som syftar till att bygga kunskap om olika tekniker och verktyg som kan användas för att identifiera mjukvarusårbarheter. Målet med studien

[1] Cybersecurity and Infrastructure Security Agency.
[2] I operationsmiljön är Försvarsmaktens definierade domäner: mark, luft, sjö, rymd och cyber [5].

är att kartlägga forskningsfältet och inrikta framtida forskning gällande identifiering av mjukvarusårbarheter. Vidare är rapporten en första del av ett flerårigt arbete med att kartlägga tekniker, verktyg och testfall. Rapporten vänder sig till läsare som har intresse av metoder för mjukvaruutveckling, tekniker och verktyg för säkerhet i mjukvara samt kvalitetssäkring av mjukvara.

Rapporten besvarar nedanstående forskningsfrågor som utgår från akademisk litteratur:

1. Vilka verktyg finns det som implementerar tekniker för att identifiera potentiella sårbarheter?
2. Vilka kategorier av sårbarheter (inkl. språk, sårbarhetstyp, m.m.) kan respektive verktyg identifiera?
3. Hur mogna är verktygen för att identifiera potentiella sårbarheter? Finns det färdiga verktyg som är praktiskt användbara i utvecklings- och produktionsmiljö?

1.2 Avgränsning

Arbetet fokuserar på verktyg producerade inom akademin. Typer av verktyg som utesluts är de som syftar till att identifiera webbsårbarheter samt verktyg för kodstöd inkluderade i utvecklingsmiljöer (exempelvis verktyg som skannar kod i realtid när en utvecklare skriver).

För att hantera den stora mängden material som ligger till grund för rapporten fokuserar den på verktyg med en hög mognadsnivå (TRL 8-9).

2 Bakgrund

En mjukvarusårbarhet är en brist i mjukvaran som kan nyttjas av en hotaktör, avsiktligt eller oavsiktligt, för att åstadkomma en oönskad händelse, som påverkar säkerheten negativt i det system som mjukvaran är del av. Säkerhet är ett relativt brett koncept som inkluderar att systemet är skyddat mot fara, exempelvis för den egna funktionen eller den information som systemet hanterar, samt att systemet inte utgör en fara för någon annan (person eller system). Att hitta och åtgärda sårbarheter i mjukvaran är därmed en del i att göra systemet säkert - en uppgift som tenderar att vara svår och tidsödande när den utförs manuellt. Ett bra verktygsstöd är önskvärt för att underlätta arbetet såväl i samband med utveckling som vid förvaltning av mjukvaran. Det finns många tekniker som verktygen kan använda för att analysera mjukvara för att identifiera sårbarheter. Detta kapitel ger en introduktion till relevanta begrepp inom området och till teknikerna som förekommer i rapporten.

2.1 Kataloger över sårbarheter

Tre kataloger inom cybersäkerhet och mjukvarusårbarheter som är relevanta för rapporten är CWE (Common Weakness Enumeration), CVE (Common Vulnerabilities and Exposures) samt NVD (National Vulnerability Database). Begreppen beskrivs nedan.

- CWE är en katalog över sårbarhetstyper på en övergripande och konceptuell nivå.[3] Det handlar om mönster eller typer av brister i design, kod eller implementation som kan leda till sårbarheter. Två exempel är CWE-416: Use after free och CWE-798: Use of Hard-coded Credentials.
- CVE är en katalog över sårbarheter.[4] När en sårbarhet har identifierats och bekräftats tilldelas den ett CVE-nummer. CVE skapar ett enhetligt och unikt sätt att referera till sårbarheter.
- NVD kan ses som en mer omfattande förlängning av CVE-katalogen.[5] Den innehåller mer data och analys kring sårbarheter, exempelvis CVSS-poäng för hur allvarlig en sårbarhet är.[6] NVD fokuserar på analys av sårbarheter som identifierats, som underlag för exempelvis riskbedömningar eller åtgärder.

2.2 Tillvägagångssätt inom mjukvarutestning

Testarbetets tillvägagångssätt påverkas av faktorer såsom mjukvarusystemets utformning, hur många system som samverkar, vilken representation av systemet som finns tillgänglig och vilken grad av insyn som finns i systemet. Med begränsad insyn i systemet följer även begränsningar i hur det går att undersöka och observera

[3] https://cwe.mitre.org
[4] https://cve.org
[5] https://nvd.nist.gov/
[6] Common Vulnerability Scoring System

funktionerna som ska testas. Nedan beskrivs tre nivåer av insyn och hur dessa spelar in i testmetodiken.

Whitebox-testning är en metod som innebär att ett system testas med stor kunskap om hur det fungerar och tillgång till koden som exekveras. Metoden kan utföras manuellt, automatiseras med ett verktyg eller en kombination av båda. Mjukvaran som testas i whitebox-testing blir en guide för hur testningen ska ske och genom att läsa koden kan man skapa testfall som ska trigga avsnitt i koden som identifieras som sårbara för manipulerad indata [8]. Exempelvis kan ett whitebox-verktyg läsa koden och hitta ett ställe som jämför längden på indata med en statisk variabel. Då genererar verktyget indata som ska träffa och testa just detta kodavsnitt.

Blackbox-testning innebär att ett system testas utan att veta vad som händer i systemet. Det enda som är känt för testaren är den indata som skickas till systemet och det utdata som kommer ut från systemet [9].

Greybox-testning är en kombination av whitebox- och blackbox-testning där det finns viss kännedom om systemets interna funktion, exempelvis genom dokumentation [10]. Till skillnad mot blackbox-testning innebär det en större möjlighet att skräddarsy testfall, men inte i lika stor utsträckning som whitebox-testning där det finns stor kännedom om systemet.

2.3 Tekniker for kodanalys

Det är svårt att göra en bra kategorisering av tekniker för kodanalys då många av dem liknar varandra, många verktyg använder flera tekniker och det ofta kräver mycket arbete att undersöka vilken teknik ett verktyg använder.

I rapporten kategoriseras tekniker med termer baserade på standarden IEC 61508-3 A.5 [11]. Denna standard innehåller en kategorisering anpassad för elektroniska och programmerbara säkerhetskritiska system, där bland annat testning och testtekniker beskrivs. Det är en beprövad standard, men den fångar inte helt utvecklingen inom forskningsområdet för de studerade åren 2014-2024. I rapporten har därför kategoriseringen kompletterats med följande fyra kategorier:

- *Statisk analys.* En kategori för statiska tekniker, motsvarande kategorin för dynamiska tekniker.
- *Mönsterigenkänning.* Kategorin samlar tekniker som baseras på maskininlärning, där mönster i källkod och binärkod används för att identifiera sårbarheter.
- *Fuzzers.* En kategori för olika tekniker som genomför många tester med varierande indata. Denna kategori särredovisas för tydlighet eftersom fuzzning är ett stort forskningsområde.
- *Hybridmetoder.* Kategorin samlar verktyg som använder flera samverkande tekniker i en kedja.

Maskininlärning är populär inom flera områden, däribland mönsterigenkänning, fuzzning och statisk analys [12], och har därför inte tagits upp som en separat kategori då den utgör en övergripande teknik. Statisk analys inkluderades i kategoriseringen eftersom många publikationer uttrycker att de använder statisk analys snarare än någon mer specifik teknik. Efter granskningen av publikationerna fanns det kategorier utan verktyg och dessa har plockats bort. De kategorier som tagits bort är gränssnittstester, statistiska tekniker och prestandatest. En förklaring till att inga verktyg hittats för kategorierna kan vara att verktygen istället klassificerats som till exempel statisk analys eller dynamisk analys. Kategoriseringen som används i rapporten presenteras i sin helhet i tabell 2.1. Nedan ges övergripande presentationer av teknikerna som identifierats i litteraturstudien.

Tabell 2.1. Rapportens kategorisering för tekniker och verktyg, baserat på IEC 61508-3, A.5.

Ursprungsterm	Rapportens term	Kommentar
-	Statisk analys	Tillägg
Dynamic analysis and testing	Dynamisk analys	
-	Fuzzning	Tillägg
Functional and blackbox testing	Funktionell analys	
Model based testing	Modellbaserad testning	
Formal verification	Formella tekniker	
-	Mönsterigenkänning	Tillägg
-	Hybridmetoder	Tillägg

2.3.1 Statisk analys

Statisk analys bygger på att systemet undersöks genom analyser av systemets binärkod eller källkod. Statiska analysmetoder innebär alltså att systemet inte körs i samband med analysen. De flesta verktyg som används för att hitta sårbarheter genom statisk analys använder någon av följande fyra tekniker: symbolisk exekvering, modellundersökning, abstrakt tolkning eller dataflödesanalys [13].

Symbolisk exekvering

Symbolisk exekvering (eng. symbolic execution) innebär att ett programs indata och variabler ersätts med abstrakta symboler som kan representera godtyckliga värden. Därigenom kan systemet analyseras utan att alla möjliga utfall i form av olika värden behöver analyseras separat. Symbolerna används även för att bestämma vilka utfall som är möjliga vid villkorliga uttryck, exempelvis vid hopp och loopar. Symbolerna och möjliga utfall i villkorliga uttryck används för att bygga upp matematiska formler över olika exekveringsvägar som indata kan ta i programmet [9]. Programkoden kan därmed kompileras till en logisk formel som sedan kan användas

tillsammans med en teorembevisare för att påvisa om det existerar en viss kombination av symbolvärden som leder till ett oönskat programtillstånd [14].

Symbolisk exekvering är en populär teknik men det kan finnas vägar som inte nås och därmed inte undersöks [12]. Symbolisk exekvering är dock en effektiv teknik för att hitta nya eller djupa delar av ett program dit det är svårt att komma med slumpartad indata och det kan ses som ett mellanting mellan statisk analys och dynamisk analys [15].

Modellundersökning

Inom modellundersökning (eng. model checking) används formell verifiering för att verifiera att ett system uppfyller dess krav. Modellundersökning använder en modell av en programs beteende baserad på programmets tillstånd och vägar mellan tillstånden. Ett exempel på modellundersökning kan vara att temporallogiska formler beskrivande systemets specifikation kontrolleras mot en finit tillståndsmaskin baserad på systemets beteende [9]. Eftersom modellundersökning är en formell teknik ligger fokus på att matematiskt verifiera modellen snarare än att skapa modellen.

Abstrakt tolkning

Abstrakt tolkning (eng. abstract interpretation) är en teknik som används för att analysera ett programs beteende utan att köra det. Den gör detta genom att skapa en förenklad version av programmets möjliga tillstånd. Den beräknar en överestimering som inkluderar både de tillstånd programmet kan nå och de som det inte kan nå. Genom att statiskt simulera programmets dynamiska beteende undersöker tekniken om programmet kan hamna i ogiltiga eller problematiska tillstånd, vilket hjälper till att identifiera potentiella säkerhetsrisker och kodfel innan programmet körs [13].

Dataflödesanalys

Dataflödesanalys (eng. data flow analysis) är en statisk testmetod som går ut på att göra programflödesanalys i kombination med att följa vilka variabler i koden som har blivit lästa eller skrivna. Dataflödesanalys kan användas för att hitta sårbarheter relaterade till exempelvis att variabler används innan de har fått värden delegerade till sig [9].

Dataflödesanalys som är specifikt inriktad på att spåra potentiell osäker data (eng. taint analysis) är också en vanlig teknik för att hitta sårbarheter. Data från osäkra källor spåras för att se hur datan rör sig genom programmet. Flödesanalys för att spåra potentiell osäker data kan användas inom både statisk analys och dynamisk analys [12].

2.3.2 Dynamisk analys

Inom dynamisk analys (eng. dynamic analysis) undersöks vad som händer i ett mjukvarusystem under körning. Systemet körs i en given, säker och övervakad miljö och data från körningen samlas in för att sedan analyseras. En svårighet med dynamisk analys är att nå och analysera alla delar av koden [15, 16]. Mjukvaran behöver vara i ett såpass färdigutvecklat stadie att den kan följa alla möjliga exekveringsvägar [9].

Det finns många olika tekniker som kan användas inom dynamisk analys. De vanligaste teknikerna är dynamisk dataflödesanalys för att spåra potentiell osäker data (eng. dynamic taint analysis), dynamisk symbolisk exekvering och fuzzning [15]. Fuzzning beskrivs under 2.3.3 eftersom det betraktas som en egen kategori i rapportens kategorisering av tekniker. Andra vanliga tekniker är analys av nätverkstrafik, minnesanalys, systemanropsanalys och spårning av systemanrop samt loggbaserad analys [16].

Dynamisk dataflödesanalys för att spåra potentiell osäker data

Dynamisk dataflödesanalys för att spåra potentiell osäker data (eng. dynamic taint analysis) liknar den statiska dataflödesanalysen för att spåra potentiell osäker data. Båda teknikerna spårar data från potentiellt osäkra källors väg genom programmet för att hitta oväntade beteenden och sårbarheter. Det som skiljer dem åt är att dynamiska dataflödesanalysen använder data genererad under körningen för få en mer tillförlitlig analys av koden. Till exempel kan dynamisk dataflödesanalys hitta vägar i kod som dynamiskt laddats in under körning. [16].

Dynamisk symbolisk exekvering

Dynamisk symbolisk exekvering (eng. dynamic symbolic execution) är en variation av symbolisk exekvering där programmet som testas först exekveras med data för att övervaka hur det beter sig. Hur programmet agerar med denna data och den väg som datan kan ta används som en utgångspunkt (ett frö) för statisk symbolisk exekvering [17].

Två vanliga tekniker för dynamisk symbolisk exekvering är symbolisk onlineexekvering och symbolisk offlineexekvering. Symbolisk onlineexekvering sparar alla tillstånd tillsammans med vägen dit. Vid varje vägval dupliceras tillståndet och varje kopia körs med de olika vägvalen. Detta ger spårbarhet och gör tekniken till en av de mest kraftfulla teknikerna för att hitta sårbarheter i binärkod [15][18]. Nackdelen med symbolisk onlineexekvering är att alla möjliga tillstånd måste sparas tillsammans med vägen dit, vilket leder till en möjligt exponentiell tillväxt av minnesanvändning.

Symbolisk offlineexekvering innebär att man vid varje vägval beräknar möjlig indata för de olika vägvalen. Därefter startas den symboliska exekveringen om från start med en av vägvalen. Detta innebär att symbolisk offlineexekvering använder sig av en iterativ process där tekniken för varje väg söker efter nya vägar. Detta medför att varje steg av körningen måste utföras minst två gånger för varje vägval [18].

Concolic execution på engelska, använder sig av statisk symbolisk exekvering parallellt med en körning av det riktiga systemet [18].[7] Tanken är att den symboliska exekveringen ska ge den konkreta körningen indata som leder till olika delar av programmet.

2.3.3 Fuzzning

Fuzzning testar ett system genom att upprepade gånger ge det indata som ändras något varje gång. Samtidigt övervakas systemet för att eventuella fel ska upptäckas. Målet är att med ogiltig, oväntad eller slumpmässig indata få programmet att krascha eller generera oväntat beteende [16].

Fuzzers kan också kategoriseras som mutationsbaserade eller genereringsbaserade beroende på hur indata genereras [19]. Mutationsbaserade fuzzers har ett givet startvärde, kallat frö (eng. seed), som modifieras för att skapa ny indata. Genereringsbaserad fuzzning baseras på en specifikation, exempelvis ett filformat, som används som utgångspunkt för att generera indata. Genereringsbaserade fuzzers kan ofta uppnå högre kodtäckning jämfört med mutationsbaserade fuzzers vilka riskerar att generera data som ofta fastnar i kontroller tidigt i programmet. En nackdel med genereringsbaserad fuzzning är att det är tidskrävande att skapa en indataspecifikation och att testfall kan missas i specifikationen.

Fuzzers kan även använda utdata från testningen. Ett exempel är täckningsstyrda (eng. coverage-guided) fuzzers som använder information om kodtäckning för att guida genereringen av indata [19]. Med täckning och täckningsgrad avses hur stor del programkoden som fuzzern testar.

2.3.4 Funktionell analys

Funktionell analys går ut på att testa funktionaliteten av ett system. Denna typ av testning fokuserar inte på själva källkoden för programmet utan snarare på hur programmet beter sig. Därför kan denna typ av testning räknas som blackbox-testning. Funktionell analys liknar dynamisk analys med skillnaden att funktionell analys används tidigare under utvecklingsprocessen för att undvika fel vid implementering och integreringen av system. Funktionell analys används framförallt under specifikation- och design-fasen av ett system [9].

2.3.5 Modellbaserad testning

Modellbaserad testning är en typ av blackbox-testning där testfall genereras utifrån en modell av ett systemet [9]. Användningsområdet för modellbaserad testning ligger i att automatiskt skapa användbara testscenarion snarare än att matematiskt kontrollera en modell. Modellen av systemet kan baseras på krav som har ställts på systemet och tekniker som kan användas för att generera testfall kan till exempel vara finita tillståndsmaskiner, markovkedjor, modellundersökning eller symbolisk exekvering.

[7] Concolic är en sammanslagning av orden concrete och symbolic.

2.3.6 Formella tekniker

Formella tekniker används för att matematiskt bevisa att en abstrakt modell av ett system uppfyller kravspecifikationen. För att formellt verifiera ett system behövs en abstrakt modell av systemet samt systemets specifikation. Denna modell användas för att matematiskt bevisa eller motbevisa att ett system uppfyller sin specifikation och klarar av all typ av indata som har specificerats för systemet [9]. Symbolisk exekvering kan använda formell verifiering för att matematiskt bevisa vilka vägar som går att nå [20]. Formella tekniker avser att bevisa ett programs korrekthet men kan ha vissa begränsningar, till exempel kan den abstrakta modellen innehålla förenklingar och det kan finnas övre begränsningar för till exempel indata och andra resurser [21].

2.3.7 Mönsterigenkänning

Mönsterigenkänning (eng. pattern recognition) är en teknik som baseras på maskininlärning [22]. För att träna en maskininlärningsmodell krävs stora mängder data och ofta används statiska tekniker för att bygga dessa dataset. Bland de vanligaste maskininlärningsmodellerna inom statisk analys finns random forest, stödvektormaskiner (SVM) och olika djupinlärningsmodeller.

2.3.8 Hybridmetoder

Hybrida metoder är metoder som använder flera olika tekniker för att hitta sårbarheter. Det är exempelvis vanligt att kombinera statiska och dynamiska tekniker. Statiska tekniker genererar ofta en stor del falska positiva och dynamiska tekniker har ofta en låg täckningsgrad av koden [23]. Hybrida tekniker kan använda både statiska och dynamiska verktyg och därmed dra nytta av båda teknikernas fördelar. Det är dock vanligt att hybridvarianten istället behåller nackdelarna från den statiska och dynamiska tekniken [16]. Även maskininlärning kombinerat med andra tekniker är vanligt. Maskininlärning kan vara effektivt i exempelvis fuzzning då modellen kan lära sig av tidigare testfall och generera mer relevanta testfall vilket kan öka täckningsgraden av koden [24].

Ett exempel på en hybridmetod är guidad fuzzning. Denna teknik använder symbolisk exekvering för att öka mängden vägar som en fuzzer hittar i ett system. Eftersom symbolisk exekvering är en kostsam process i form av tid och manuellt arbete försöker guidad fuzzning minska användningen av den symboliska analysen.

3 Metod

Litteraturstudien genomfördes utifrån den metod som Kitchenham m.fl. tagit fram för systematiska litteraturstudier inom mjukvaruteknik [25]. Denna metod har blivit något av en standardmetod för litteraturstudier inom forskningsområdet.

Rapportens valda metod består av följande huvudmoment:

1. Litteratursökningar i databaser med hjälp av utvalda sökord följt av gallring av uppenbart ointressanta träffar utifrån titlar.
2. Granskning av sammanfattningar för att bedöma relevansen av återstående publikationer och därigenom gallra bort mindre relevanta träffar.
3. Snöbollning via referenslistor för att hitta eventuella publikationer som missats vid sökningarna.
4. Granskning av fulltext för att mognadsbedöma verktyg som presenteras i publikationerna.
5. Tematisk analys av publicerade verktyg som bedömts ha högst mognadsgrad.

En illustration över metoden återfinns i figur 3.1. Följande avsnitt beskriver respektive huvudmoment i mer detalj.

3.1 Litteratursökning

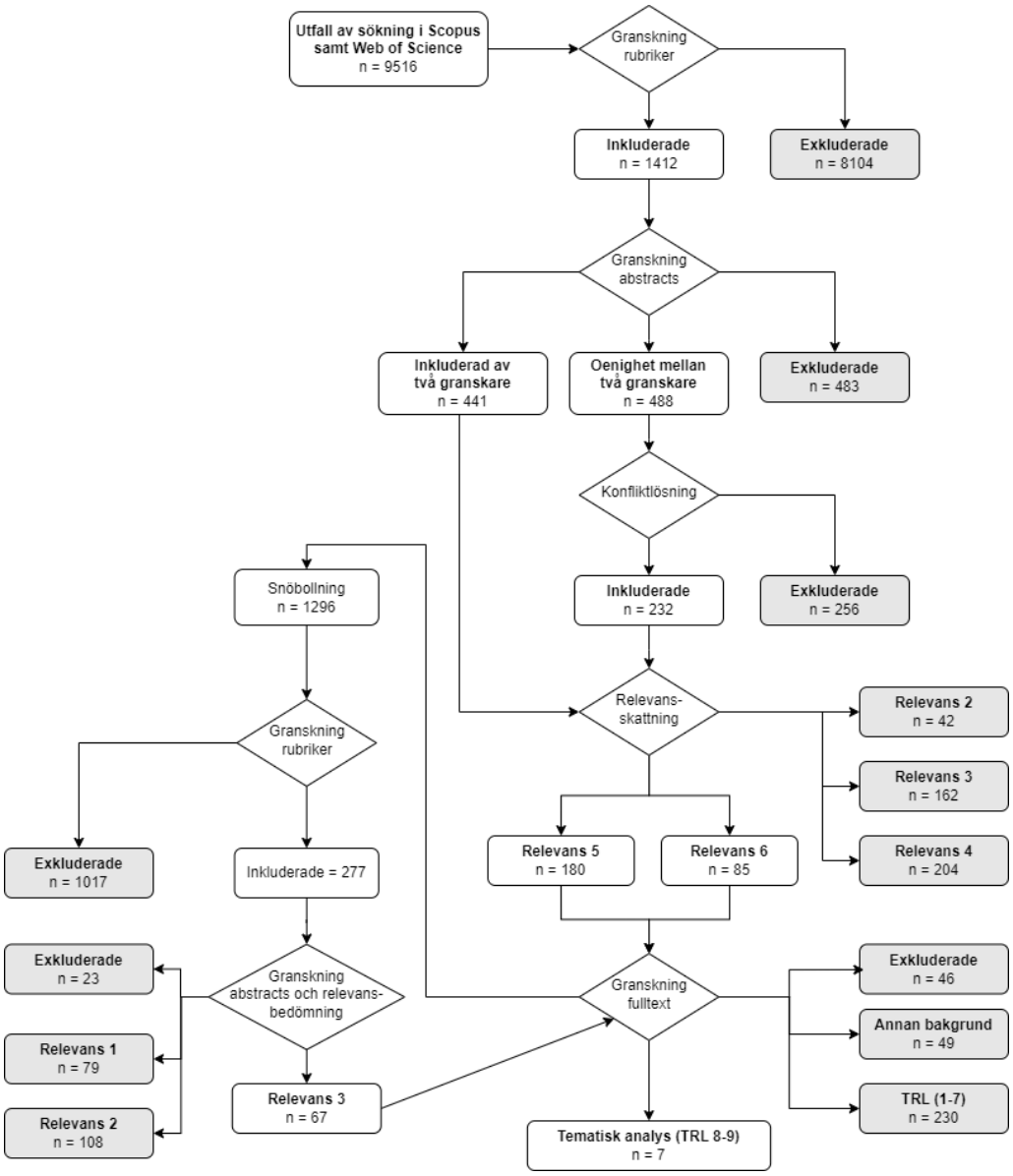
En initial ansats för söktermer diskuterades under en workshop med rapportens författare. Pilotsökningar genomfördes i Scopus för att få en bild av sökresultatet. Sökningarna resulterade i en stor mängd irrelevanta publikationer, vartefter söktermerna förfinades vid en uppföljande workshop.

Litteratursökningen genomfördes i Scopus och Web of Science med fyra kategorier av söktermer som redovisas i tabell 3.1. Sökningen genomfördes med boolesk kombination, vilket innebär att varje sökresultat innehöll minst ett ord från varje kolumn.

Tabell 3.1. De fyra kategorier av söktermer som användes vid litteratursökningen. Asterisk (*) anger valfri fortsättning på ordet.

Söktermer			
Måltavla	**Sårbarhet**	**Aktivitet**	**Verktyg**
Software Source code Binary	Vuln* Weakness Bugs	Detection Analysis	Technique Tool

Resultaten avgränsades till åren 2014 – 2024, till publikationer på engelska samt till vetenskapliga tidsskriftsartiklar och konferensartiklar. Sökningen avgränsades också till kategorierna Computer Science Information Systems, Computer Science Theory Methods samt Computer Science Software Engineering.



Figur 3.1. Övergripande beskrivning av stegen i litteratursökningen och granskningsprocessen.

Den initiala litteratursökningen resulterade i 9 516 publikationer efter rensning av dubbletter.

Utfallet granskades på titelnivå för att rensa materialet från uppenbart irrelevanta publikationer. Granskningen genomfördes av tre personer som initialt diskuterade rubriker för de första 100 publikationerna. Därefter genomfördes rensningen i det resterande materialet uppdelat i var sin tredjedel. Vid tveksamheter inkluderades publikationen hellre än att den exkluderades.

Efter gallring på titelnivå exkluderades 8 104 publikationer. I huvudsak handlar de exkluderade publikationerna om områdena webbsårbarheter, hantering av buggrapporter, kodkloning, smarta kontrakt för kryptovalutor, optimering av kodprestanda, granskning av loggar, intrångsdetektion och övervakning av nätverk.

Det slutliga utfallet av titelgranskningen bestod av 1 412 publikationer för fortsatt granskning.

3.2 Granskning av sammanfattningar

Sökresultatets 1 412 sammanfattningar relevansgranskades i två steg med syfte att identifiera de publikationer som var mest relevanta för fulltextgranskning.

I ett första steg bedömdes enbart om publikationen skulle inkluderas eller exkluderas baserat på kriterierna i tabell 3.2.

Tabell 3.2. Kriterier för inkludering och exkludering vid granskning av publikationernas sammanfattningar.

Granskningskriterier	
Kriterier för inkludering	**Kriterier för exkludering**
Litteraturstudier, taxonomier och kunskapssammanställningar. Beskriver tekniker och verktyg som identifierar mjukvarusårbarheter. Beskriver tekniker och verktyg på ett sätt som är reproducerbart/ tillämpbart. Redovisar tester av tekniker och verktyg.	Ej relevant för studiens forskningsfrågor eller utifrån fastställda avgränsningar. Fokuserar på utvecklarbeteende, kodoptimering och kodningsstöd. Teoribildningar utan praktiska resultat. Smala, systemspecifika verktyg utan bredare tillämpningsområde. Fokus på förberedelser och planering.

Granskningen genomfördes av tre personer, där materialet delades upp så att varje sammanfattning lästes av två granskare. De 50 första granskningarna genomfördes gemensamt för att diskutera kriterierna innan tillämpning. Därefter gjordes varje

bedömning separat. När två granskare gjort olika bedömning avseende inkludering diskuterades bedömningen med den tredje granskaren som gjorde en utslagsgivande bedömning.

Likvärdiga bedömningar gjordes för de flesta publikationer, men i 488 fall skiljde de sig åt. Av granskningarna där bedömningen skiljde sig åt inkluderades 232 för fulltextgranskning, medan 256 exkluderades. Sammanlagt inkluderades 673 publikationer.

I ett andra steg skattades publikationens relevans på en tregradig skala utifrån dess sammanfattning:

1. Tveksam relevans, eller endast en delmängd av publikationen verkar vara av intresse för studien.
2. Relevant, bedöms vara användbar för studien och uppfyller minst ett av kriterierna för inkludering.
3. Hög relevans, helt i linje med studiens intressen. Möter flera av kriterierna för inkludering.

Skattningen genomfördes av tre personer, där varje publikation granskades av två personer oberoende av varandra. Summan av relevansskattningarna, som finns på en skala från 2 – 6, användes som kriterium för dess relevans. Fördelningen av publikationernas skattade relevans framgår av tabell 3.3.

Tabell 3.3. Fördelning av relevansskattningar baserad på publikationernas sammanfattningar. Skattad relevans är summan av två granskares bedömning på en tregradig skala.

Skattad relevans	**6**	**5**	**4**	**3**	**2**
Antal publikationer	85	180	204	162	42

De 265 publikationer med högst skattad relevans, skattning fem till sex, gick vidare för fulltextgranskning.

3.3 Snöbollning

En automatiserad snöbollning i ett steg bakåt genomfördes baserat på publikationerna med skattning fem till sex. Genom databasen Open Citation sammanställdes refererade arbeten från 233 publikationer.[8] För 32 publikationer saknades doi-nummer och dessa har därför uteslutits ur snöbollningsarbetet.

Snöbollningen resulterade i 2 690 publikationer efter att dubbletter exkluderats. Utöver detta exkluderades även 252 publikationer som redan granskats under arbetet med litteraturstudien. När utfallet reducerats till åren 2014-2024 återstod 1 296 publikationer. För att rensa materialet från uppenbart irrelevanta publikationer samt dubbletter granskades publikationerna manuellt på rubriknivå av en person. Efter rubrikgranskningen återstod 277 publikationer.

[8] https://opencitations.net/

En granskning av publikationernas sammanfattningar genomfördes av en person. Samma inkluderings- och exkluderingskriterier som tidigare användes, liksom samma tregradiga skala som tidigare för relevansskattning. 23 publikationer exkluderades utifrån exkluderingskriterierna. Skattad relevans för återstående publikationer redovisas i tabell 3.4.

Tabell 3.4. Fördelning av relevansskattningar under snöbollningen baserad på publikationernas sammanfattningar. Skattad relevans är resultatet av en granskares bedömning på en tregradig skala.

Skattad relevans	**3**	**2**	**1**
Antal publikationer	67	108	79

67 publikationer med relevansskattning tre inkluderades för fulltextgranskning. För att undersöka varför de inte kom med i den ursprungliga litteratursökningen jämfördes ett urval av dem mot använda söktermer. Frånvaron av enskilda söktermer, till exempel *technique* och *tool*, visade sig vara skälet till att publikationerna inte hade inkluderats tidigare.

3.4 Granskning av fulltext

Litteratursökningen och snöbollningen gav tillsammans 332 publikationer som bedömts som relevanta för fulltextgranskning. Denna granskning skedde med hjälp av ett granskningsprotokoll vars struktur baseras på Granåsen m.fl. [26]. Varje publikation granskades av en granskare.

Vid fulltextgranskning exkluderades publikationer där det visade sig att innehållet inte motsvarade vad sammanfattningen beskriver. För detta användes samma inkluderings- och exkluderingskriterier som tidigare beskrivits i tabell 3.2.

Publikationerna kategoriserades utifrån de tekniker och verktyg som beskrivs, enligt taxonomin i tabell 2.1.

För att bedöma mognadsgraden för de tekniker och verktyg som beskrivs i publikationerna genomfördes en mognadsskattning. Grunden för skattningen var en skala för TRL, Technology Readiness Level [27]. För rapporten används en svensk tolkning, beskriven i tabell 3.5, som tidigare använts av KTH för arbete i Horizon 2020 Work Programmes.[9] Sju av publikationerna bedöms ha en TRL-nivå på 8 eller 9.

Under arbetet identifierades 88 publikationer som inte gick att TRL-skatta, eftersom de inte beskriver tekniker eller verktyg. Av dem exkluderades 39 publikationer helt, baserat på studiens inkluderings- och exkluderingskriterier. Resterande 49 publikationer bedömdes som potentiellt bakgrundsmaterial för rapporten. Utöver detta identifierades sju publikationer som dubbletter och därmed exkluderades de; totalt exkluderades 46 publikationer.

[9] https://intra.kth.se/forskning/extern-forskningsfin/internationell-finan/eu-finansiering/cross-cutting-issues/trl-technology-readiness-levels-1.406254

Tabell 3.5. Kriterier för bedömning av mognadsgrad, TRL, Technology Readiness level.

TRL	Beskrivning
9	Systemet beprövat i operationell miljö
8	Systemet komplett och bekräftat
7	Systemprototyp demonstrerad i operationell miljö
6	Teknologi demonstrerad i relevant miljö
5	Teknologi validerad i relevant miljö
4	Teknologi bekräftat i lab
3	Experimentellt bevisat koncept
2	Teknologikoncept finns formulerat
1	Basforskning

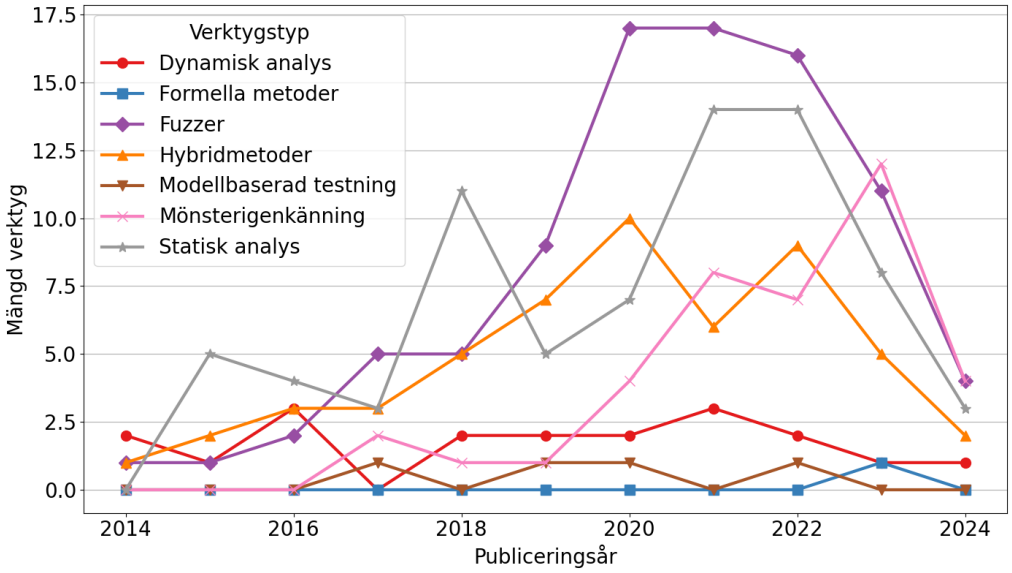
3.5 Tematisk analys

En tematisk analys genomfördes av de sju publikationer med TRL-nivå 8-9 för att identifiera ytterligare teman och resonemang i publikationerna som inte fångades upp av granskningsprotokollet. Varje publikation analyserades initialt av två av författarna och vid efterföljande iterationer förfinades och grupperades identifierade teman.

Resultaten från den tematiska analysen identifierade teman i publikationerna som sedan användes för att inrikta djupare analyser och diskussioner i rapporten. Analysarbetet genomfördes i mjukvaruverktyget NVivo.

4 Resultat

Utifrån publikationerna som fulltextgranskades går det att observera flera trender över de senaste tio åren. Publikationerna domineras av fuzzers och statiska analysverktyg även om mönsterigenkänning ökat mycket sedan 2019. Fuzzers och statiska analysverktyg har också en tydlig topp mellan 2020 och 2022 men minskade tydligt till 2023. När det kommer till hybridmetoder är trenden ungefär som ett medelvärde för kategorierna som hybridmetoder oftast består av: dynamisk analys, statisk analys, fuzzers och mönsterigenkänning.



Figur 4.1. Publikationstrender per verktygstyp för samtliga TRL-nivåer. Notera att år 2024 endast inkluderar delar av året.

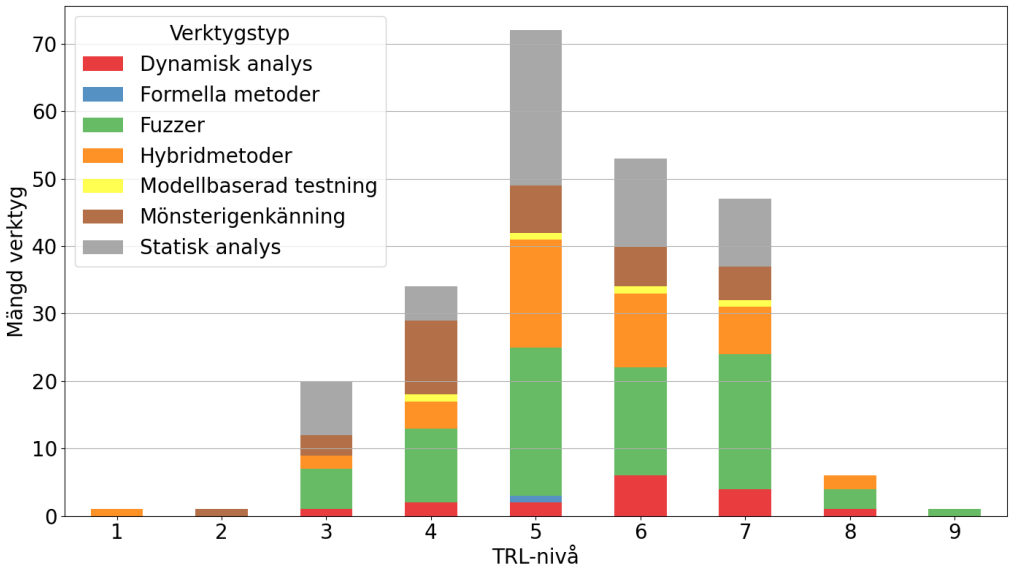
Fördelningen av den skattade mognadsnivån för tekniker och verktyg som presenteras i publikationerna framgår av tabell 4.1.

Tabell 4.1. Fördelning av mognadsbedömningar baserad på fulltextgranskning av publikationerna.

Skattad mognad	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Antal publikationer	1	6	47	53	74	34	20	1	1

I figur 4.2 illustreras studiens bedömning av verktygens mognadsgrad samt hur olika tekniker fördelar sig över TRL-nivåerna. Den mognadsnivå som har bedömts som vanligast är TRL-nivå 5, vilket innebär att tekniken som verktyget nyttjar validerats eller demonstrerats i en relevant miljö. Ett exempel på ett verktyg som under fulltextgranskningen bedömdes vara på TRL-nivå 5 beskrivs i en publikation av Partenza m.fl. [28]. De beskriver en experimentell studie med syfte att skapa ett verktyg baserat på maskininlärning för att hitta sårbarheter. Verktyget presterade

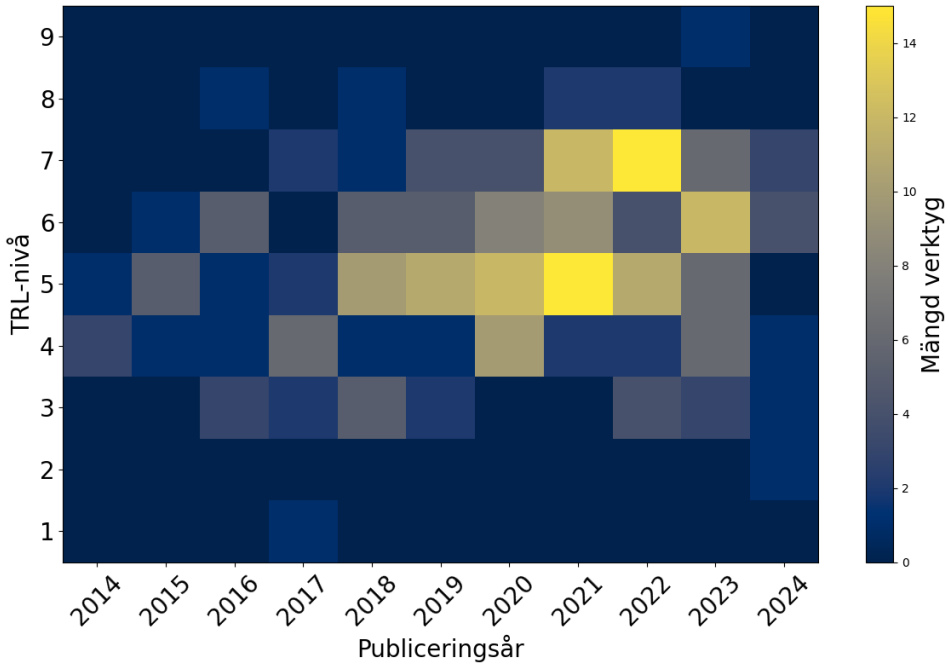
dåligt när det testades på nya befintliga sårbarheter, ur OWASP dataset,[10] vilket troligtvis berodde på ett bristande träningsdataset. Verktyget har inte heller jämförts mot andra liknande verktyg. Ett annat verktyg, AFLSmart, är beskrivet i en publikation av Pham m.fl. [29] och har under fulltextgranskningen bedömts med TRL-nivå 8, vilket innebär att verktyget är komplett och bekräftat.[11] AFLSmart bygger vidare på den väletablerade fuzzern AFL och testas utförligt mot flera välkända fuzzers som AFL, AFLFast[12], och VUZZER med en uppsättning produktionsmjukvaror. Testerna visar att AFLSmart hittar fler sårbarheter i publikationens utvärdering och att det även har hittat nolldagssårbarheter i produktionsmjukvara. Få andra verktyg når upp till en hög mognadsgrad enligt studiens bedömningskriterier, där endast sju verktyg bedömdes vara på TRL-nivåerna 8-9.



Figur 4.2. Mognadsgrad per verktygstyp

I figur 4.3 presenteras hur många verktyg som uppnår en viss mognadsgrad per år för perioden 2014–2024. Innan år 2018 är det generellt sett en liten mängd verktyg som, oavsett mognadsgrad, har inkluderats i fulltextgranskningen. Antalet har sedan ökat kraftigt från år 2020.

[10] https://owasp.org/www-project-benchmark/
[11] https://github.com/aflsmart/aflsmart
[12] https://github.com/mboehme/aflfast



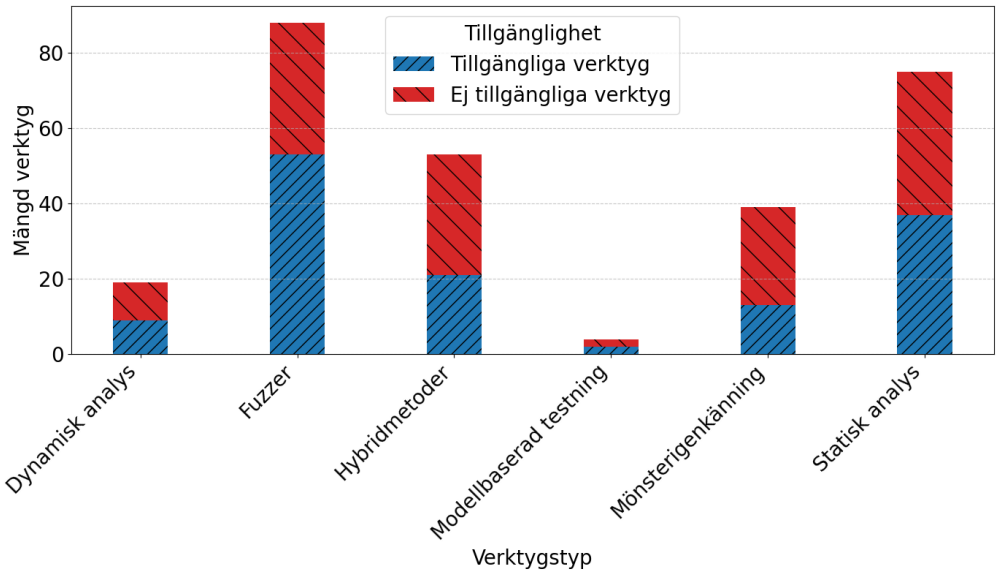
Figur 4.3. Hur många verktyg som publicerades under ett år kategoriserat efter mognadsgrad.

4.1 Verktygstyper

Observeras resultatet i figur 4.2 utifrån vilka verktygstyper som når de olika TRL-nivåerna är fuzzers högt representerade bland TRL 8 och 9. Även för nivå 7 utgör fuzzers med god marginal den största gruppen verktyg. I figur 4.4 visas antalet tillgängliga verktyg för respektive verktygstyp baserat på verktyg med TRL 1-9. Med tillgängligt avses att verktyget är öppen källkod och går att få tag på, till exempel på GitHub. Antalet publicerade fuzzers som är tillgängliga överstiger klart alla andra kategorier även om statiska tekniker är vanligt förekommande. Hybridmetoder och mönsterigenkänning förekommer i liknande antal medan dynamiska analysverktyg och modellbaserad testning endast förekommer i ett fåtal verktyg. Jämfört med taxonomin i tabell 2.1 är det flera tekniker som inte förekommer i artiklarna som granskades: statistiska tekniker, funktionell analys, prestandatest, gränssnittstester och formella tekniker.

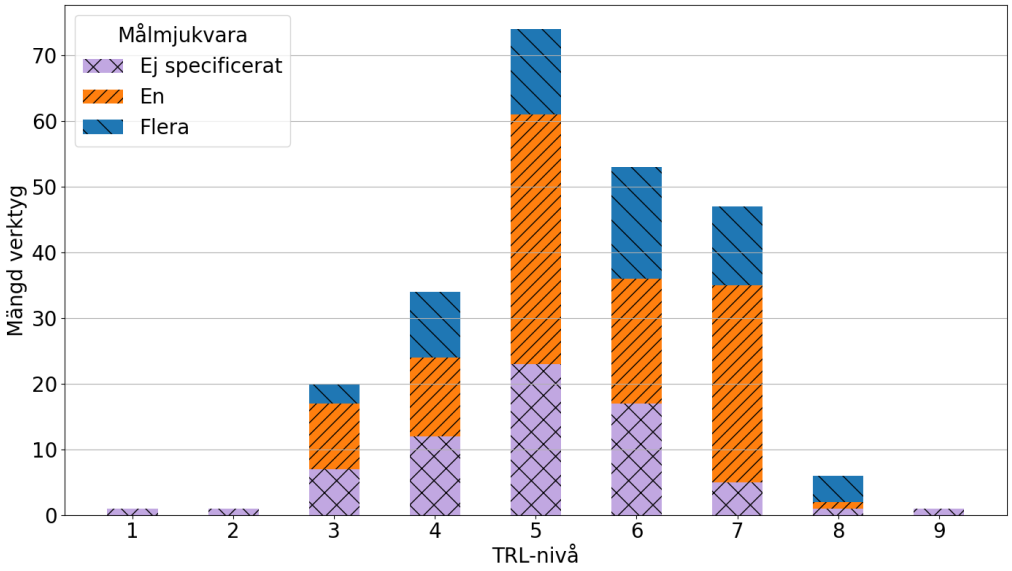
4.2 Verktygens målmjukvaror

Figur 4.5 illustrerar antalet verktyg som inriktar sig på en eller flera målmjukvaror för de olika TRL-nivåerna. Målmjukvara refererar till den typ av mjukvara som verktyget analyserar. En målmjukvara kan vara ett programspråk eller ett typ av system, till exempel firmware, binärer eller androida applikationer. Av de totalt 237 artiklarna är det 110 som riktar sig mot endast en målmjukvara. Ytterligare 59 identifierade artiklar riktar sig mot flera olika målmjukvaror, varav 49 enbart fokuserar på kombinationen C och C++. Resterande 68 artiklar återfinns främst på lägre TRL-nivåer och beskriver inte tydligt vilken målmjukvara de riktar sig mot. I



Figur 4.4. Mängd artiklar som presenterar tillgängliga respektive otillgängliga verktyg för varje verktygstyp. Data utgår från artiklar med samtliga TRL-nivåer.

bilaga B finns en mer detaljerad redovisning över TRL-nivå, målmjukvara och år för respektive publikation.

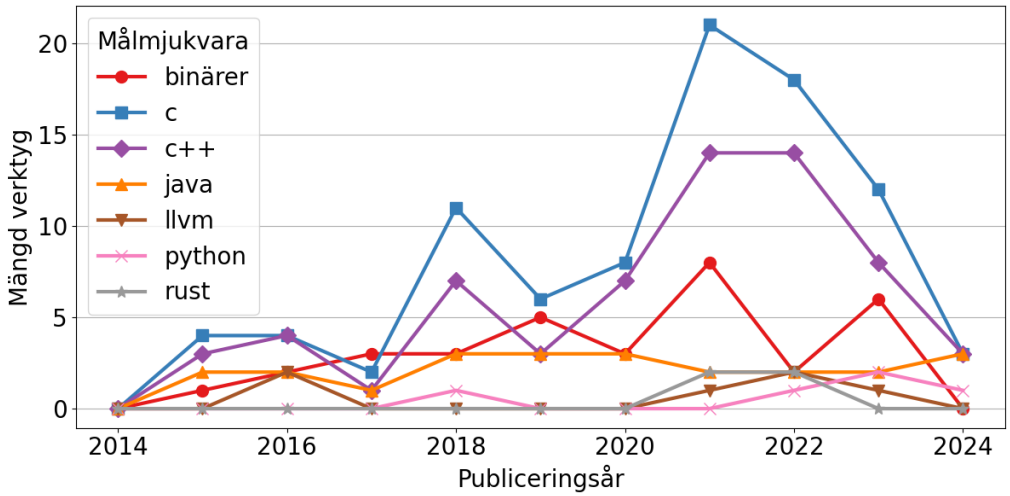


Figur 4.5. Antal verktyg som inriktar sig på en eller flera målmjukvaror i de olika TRL-nivåerna.

Majoriteten av verktygen som ingick i publikationerna som fulltextgranskades fokuserade på C, C++, Java eller binärkod. Andra programspråk och målprogramvaror förekommer men är få i antal och utspridda över olika verktygstyper. C och C++ rankas relativt högt avseende språkanvändning men är inte de mest populära [30, 31]. Vidare finns det en tydlig diskrepans mellan vilka programmeringsspråk som är vanligast att använda och vilka målmjukvaror som analysverktygen inriktar sig mot, vilket kan observeras genom att jämföra data från

Cass [30] och Gewirtz [31] med figur 4.6. Python är näst intill obefintligt bland publikationerna trots att det varit ett av de mest populära programmeringsspråken i flera års tid. En förklaring till överrepresentation av C och C++, till skillnad från Python, kan vara att minnesrelaterade sårbarheter är mer vanligt förekommande i dessa språk eftersom de saknar inbyggda skyddsmekanismer för minnessäkerhet [32]. I denna studie har betydligt färre analysverktyg identifierats för minnessäkra programmeringsspråk såsom Python, C# och Rust.

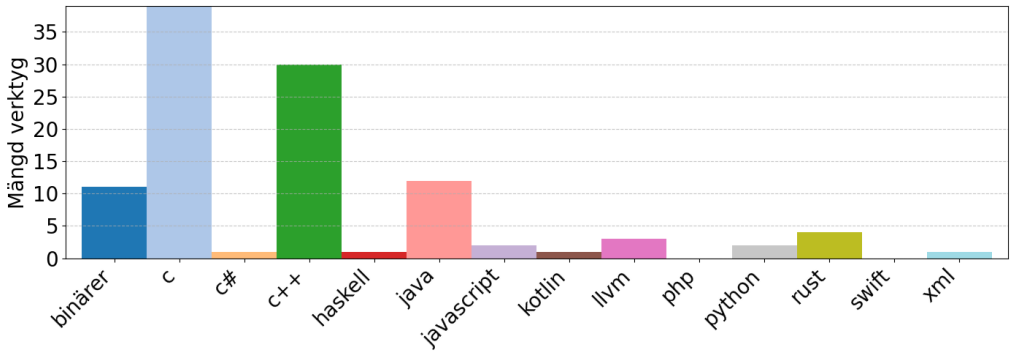
En delmängd av verktygen i studien fokuserar inte på programmeringsspråk utan inriktar sig exempelvis på firmware, drivrutiner, IoT och operativsystem såsom Mac OS och Android. Efter C och C++ kommer binärkod som tredje största målmjukvara.



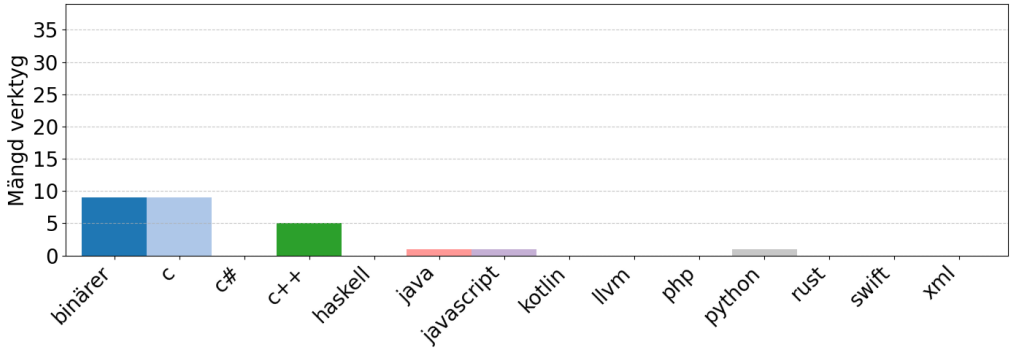
Figur 4.6. Trender för målmjukvara.

I figur 4.7 till 4.11 presenteras mängden verktyg som inriktar sig mot respektive målmjukvara för de olika verktygstyperna. Ingen distinktion mellan verktyg som hanterar en eller flera målmjukvaror har gjorts i dessa figurer. En notering utifrån figurerna är att det finns en generell popularitet för C, C++, binärer och Java som målmjukvaror. Bland dessa målmjukvaror är det C och C++ som är dominerande.

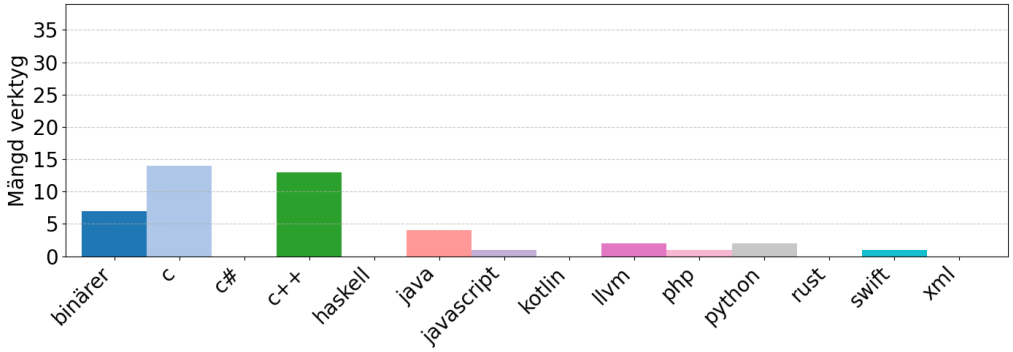
En observation kan göras kring resultaten i figur 4.7 och 4.8. Antalet identifierade fuzzerverktyg i studien är fler än verktyg för statisk analys. Publikationer om statisk analys-verktyg anger tydligare målmjukvara, vilket gör att det i figurerna kan framstå som att antalet fuzzers är färre.



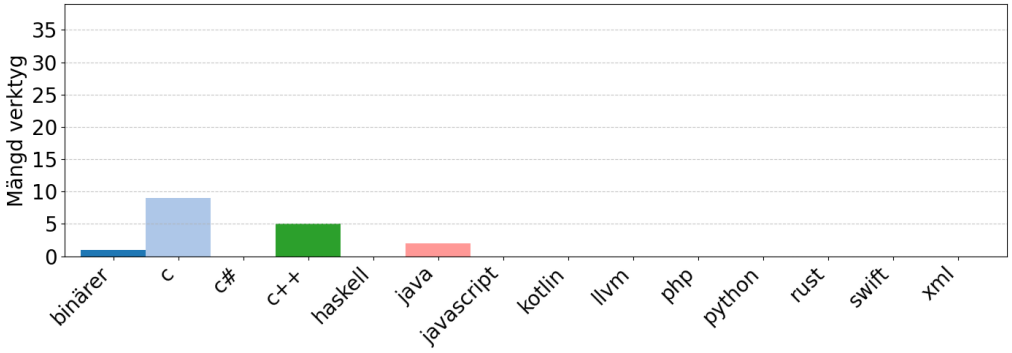
Figur 4.7. Mängd statiska analysverktyg som inriktar sig på respektive målmjukvara. Data utgår från artiklar med samtliga TRL-nivåer.



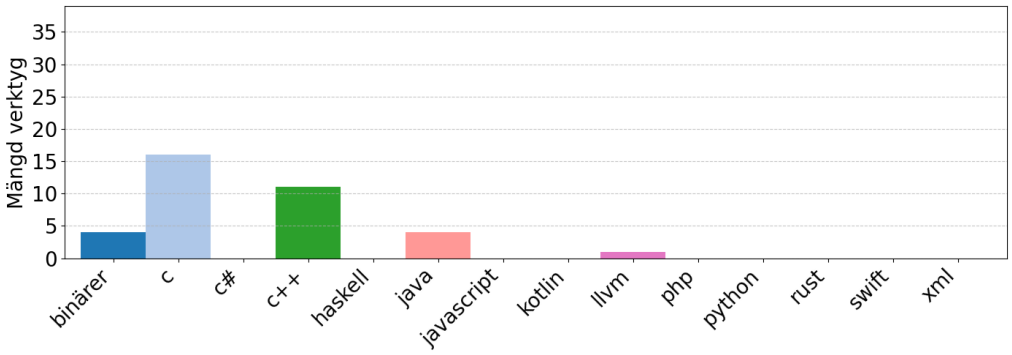
Figur 4.8. Mängd fuzzer-verktyg som inriktar sig på respektive målmjukvara. Data utgår från artiklar med samtliga TRL-nivåer.



Figur 4.9. Mängd mönsterigenkänningsverktyg som inriktar sig på respektive målmjukvara. Data utgår från artiklar med samtliga TRL-nivåer.



Figur 4.10. Mängd dynamiska analys verktyg som inriktar sig på respektive målmjukvara. Data utgår från artiklar med samtliga TRL-nivåer.



Figur 4.11. Mängd hybridverktyg som inriktar sig på respektive målmjukvara. Data utgår från artiklar med samtliga TRL-nivåer.

4.3 Verktygen med högst mognadsgrad

Ur artiklarna identifierades sju verktyg som bedömdes vara av hög mognadsgrad, det vill säga på TRL nivå 8–9. Dessa verktyg beskrivs i tabell 4.2 tillsammans med de CWE:er och sårbarhetstyper som verktygen fokuserar på. Verktygen kan delas in i kategorierna fuzzer, symbolisk exekvering[13] och statisk analys baserat på vilken teknik som verktygen nyttjar. De olika kategorierna och verktygen beskrivs nedan. Några av verktygen använder flera tekniker, dessa har kategoriserats som den teknik som framförallt används. I tabell 4.2 anges teknikerna som verktygets författare uppger är verktygens huvudsakliga tekniker.

Symbolisk exekvering är den mest använda tekniken bland verktygen. Inception, Angr och Sydr använder framförallt symboliskt exekvering men tekniken används även av Fuzzware och Goshawk under någon fas i verktygens analysprocesser. Gemensamt för många av verktygen är att analysen utförs stegvis. Goshawk använder till exempel en maskininlärningsmodell för att hitta potentiellt intressant kod som sedan analyseras statiskt vilket reducerar mängden falska positiva resultat. För att hantera olika format och språk analyserar både Polyfuzz och Inception en intermediärrepresentation av koden. En intermediärrepresentation användas vanligtvis som ett mellansteg när kod kompileras och kan ses som ett mellanting mellan källkod och maskinkod [33]. Fuzzware lyfter ut mjukvaran till en emulator. Sydr kombinerar två olika verktyg varav ett analyserar koden och det andra exekverar programmet.

4.3.1 Fuzzers

Den vanligaste tekniken i litteraturgranskningen var fuzzers, vilket återspeglas även bland de verktygen som bedömts ha högst mognadsgrad. Följande är en sammanfattning av fuzzer-verktygen som blev bedömda som nivå 8 eller 9 på TRL-skalan.

AFLSmart är ett verktyg som presenterades av Pham m.fl. i en publikation från 2018 och baseras på den väletablerade fuzzern AFL [29]. Verktyget använder sig av författarnas nya strategi som de kallar smart greybox–fuzzning. Smart greybox–fuzzning har som mål att konstruera bättre indata till fuzzern vilket ska resultera i att fuzzern spenderar mer tid på att generera indata som faktiskt kommer att ta sig in djupare i programmet istället för att det fastnar eller förkastas av tidiga kontroller i mjukvaran. Författarna presenterar resultat där AFLSmart har hittat 42 nolldagssårbarheter där 22 av dem fått CVE-nummer

[13] Både statisk symbolisk exekvering och dynamisk symbolisk exekvering.
[14] Sårbarheter som verktygets författare uttrycker att verktyget kan hitta.
[15] Om verktyget hittat nya sårbarheter som blivit tilldelade CVE:er nämns det oftast i publikationen eller verktygets Git. CWE:er från CVE:erna är angivna här.
[16] Författaren nämner fem CVE:er som verktyget hittat men information om dessa går inte att hitta. CVE-numren är märkta som reserverade i databasen CVE List.
[17] Verktyget hittar 47 nya use-after-free-buggar och 45 nya double-free-buggar men inga CVE nämns.
[18] Sårbarheter som verktygets författare uttrycker att verktyget kan hitta.
[19] Om verktyget hittat nya sårbarheter som blivit tilldelade CVE:er nämns det oftast i publikationen eller verktygets Git. CWE:er från CVE:erna är angivna här.
[20] CWE:er där verktyget hittar alla eller de flesta redan kända sårbarheterna i datasetet.

Tabell 4.2. Verktyg med TRL 8-9

Verktyg	Teknik	Sårbarhetstyper[14]	CWE[15]
AFLSmart	Fuzzning (greybox)	Assertion Failure, Divide-by-Zero, Heap/Stack Overflow, Null Pointer Reference	CWE-20: Improper Input Validation, CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, CWE-125: Out-of-bounds Read, CWE-129: Improper Validation of Array Index, CWE-190: Integer Overflow or Wraparound, CWE-369: Divide By Zero, CWE-476: NULL Pointer Dereference, CWE-617: Reachable Assertion, CWE-787: Out-of-bounds Write
Angr	Symbolisk exekvering	Buffer overflow, Saknas tydlig information	Saknas tydlig information
PolyFuzz	Fuzzning (greybox)	Saknas tydlig information	-[16]
Goshawk	Statisk analys	Use-after-free, Double-free	-[17]
Fuzzware	Dynamisk Symbolisk exekvering, täckningsstyrd fuzzning	Buffer out of Bounds, Size field integer underflow corruption of memory, Buffer Overflow	CWE-120: Buffer Copy without Checking Size of Input (’Classic Buffer Overflow’), CWE-121: Stack-based Buffer Overflow, CWE-122: Heap-based Buffer Overflow, CWE-125: Out-of-bounds Read, CWE-130: Improper Handling of Length Parameter Inconsistency, CWE-191: Integer Underflow (Wrap or Wraparound), CWE-476: NULL Pointer Dereference, CWE-665: Improper Initialization, CWE-680: Integer Overflow to Buffer Overflow, CWE-703: Improper Check or Handling of Exceptional Conditions, CWE-787: Out-of-bounds Write, CWE-843: Access of Resource Using Incompatible Type (’Type Confusion’)

Verktyg med TRL nivå 8-9

Verktyg	Teknik	Sårbarhetstyper[18]	CWE[19]
Sydr	Dynamisk Symbolisk exekvering	Null Pointer Dereference, Division by Zero, Out-of-Bounds Access, Integer Overflow, Buffer Overflow	CWE-121: Stack-based Buffer Overflow, CWE-122: Heap-based Buffer Overflow, CWE-124: Buffer Underwrite (’Buffer Underflow’), CWE-126: Buffer Over-read, CWE-127: Buffer Under-read, CWE-190: Integer Overflow or Wraparound, CWE-191: Integer Underflow (Wrap or Wraparound), CWE-194: Unexpected Sign Extension, CWE-369: Divide By Zero, CWE-680: Integer Overflow to Buffer Overflow[20]
Inception	Symbolisk exekvering	Division by Zero, Null Pointer Dereference, Use-After-Free, Free Memory Not on Heap, Heap-Based Buffer Overflow, Double-free	Saknas tydlig information

tilldelade. Av dessa nolldagssårbarheter kunde även grundläggande AFL identifiera 16 av dem. AFLSmart är fritt tillgängligt på GitHub och uppdaterades senast 2020.

Fuzzware är ett hopslaget namn från de två delar som verktyget täcker: fuzzer och firmware. Verktyget är gjort av Scharnowski m.fl. och beskrivs i en publikation från 2022 [34]. Verktyget testar inte den inbyggda mjukvaran i sin tänkta hårdvara utan gör detta istället i en avskalad emulator. Verktyget fokuserar på att skapa relevanta indatavärden baserat på vilka hårdvarufunktioner som används av den inbyggda mjukvaran. Fuzzware hittade under sin utvärdering 15 nya sårbarheter där 12 av dessa fått CVE-nummer tilldelade.

Fuzzware finns tillgängligt på GitHub och verktyget får fortfarande små sporadiska uppdateringar.[21]

PolyFuzz är en fuzzer specifikt utvecklad för att hantera flerspråkiga program och kodbaser. Skaparna av PolyFuzz, Li m.fl., menar i sin publikation från 2023 att majoriteten av fuzzer-verktyg endast fokuserar på ett programspråk vilket gör dem mindre effektiva när det kommer till kodbaser och program som innehåller flera programspråk [35].

PolyFuzz är i grunden en greybox-fuzzer som under en komplex förberedelsefas förbereder programmet för fuzzning och skapar relevant indata. Verktyget kan dock inte hantera alla kombinationer av programspråk utan är utvecklat och testat för C, Java och Python. Författarna menar dock att verktyget går att utöka för andra programspråk med relativt liten arbetsinsats [35]. För att hantera olika semantiker och språkstrukturer har PolyFuzz en egen specialgjord intermediärrepresentation.

I författarnas egen utvärdering av Polyfuzz hittades 14 nya sårbarheter som fått totalt fem CVE-nummer tilldelade [35].

PolyFuzz har öppen källkod och finns att hitta på GitHub. Arbetet på verktyget startades under 2020 och har uppdaterats ett par gånger per år sedan dess.[22]

4.3.2 Symboliska exekveringsverktyg

Symbolisk exekvering var inte en av de övergripande kategorierna beskrivna i tabell 2.1 men utgör en stor andel av verktygen som hade hög mognadsgrad. Följande är en sammanfattning av de symboliska exekveringsverktyg som blev bedömda som 8 på TRL-skalan.

Angr är en plattform för analys av binärer som presenterades av Shoshitaishvili m.fl. i en publikation från 2016 [36]. Plattformen erbjuder många olika funktioner, till exempel kontrollflödesanalys, återskapande av krascher och generering av exploits. Huvudsakligen erbjuder Angr sårbarhetsanalys genom olika symboliska exekveringsmetoder. Bland annat har Angr ursprungligt stöd för dynamisk symbolisk exekvering och fuzzning med stöd av symbolisk exekvering. Skaparna har konstruerat plattformen för att underlätta och

[21] https://github.com/fuzzware-fuzzer/fuzzware
[22] https://maartengr.github.io/PolyFuzz/

möjliggöra tillägg av andra analysmetoder. Tanken är att flera metoder ska kunna jämföras på ett likvärdigt sätt.

Verktyget har en egen hemsida och all kod finns tillgänglig på GitHub.[23] Angr har uppdateras kontinuerligt sedan 2015.

Inception är ett ramverk som kan analysera inbyggd mjukvara i ett helhetsperspektiv. Inceptionramverket presenterades av Corteggiani m.fl. i en publikation från 2018 [37]. Författarna menar att ett inbyggt system måste undersökas i ett helhetsperspektiv, det vill säga analysera källkod, assemblerkod och binärkod tillsammans samtidigt som funktionalitet från den tänkta hårdvaran kan påverka programmet. Inception gör detta genom att översätta källkoden till LLVM:s intermediärrepresentation och lyfta assemblerkod och binärkod till samma format. LLVM är ett projekt som bland annat skapar en modulär och återanvändbar kompilator. För att göra detta använder projektet en egen intermediärrepresentation [38].

Som analysverktyg använder sig Inception av KLEE som är ett välkänt verktyg för dynamisk symbolisk exekvering. Skaparna har dock utökat KLEE för att hantera minne och avbrottsrutiner (eng. interrupts) från hårdvaran samtidigt som analys körs på den översatta LLVM-koden. Författarna rapporterar att Inception hittade åtta olika krascher och två nya sårbarheter.

Verktyget är tillgängligt på GitHub och uppdaterades senast 2022.[24]

Sydr är ett verktyg för dynamisk symbolisk exekvering, presenterat av Vishnyakov m.fl. i en publikation från 2020 [39]. Sydr är uppbyggt av två andra verktyg, Trition[25] och DynamoRIO.[26] Trition bidrar med analysdelen av Sydr medan DynamoRIO tillhandahåller exekveringen av programmet som ska analyseras. Författarnas bidrag till verktyget är förbättringar av den symboliska exekveringen. De utgick från den senaste forskningen och använde teknikerna som var lovande för att förbättra symbolisk exekvering.

Vishnyakov m.fl. har presenterat flera uppföljningsstudier på Sydr. Som exempel publicerades en rapport 2021 [40] som går igenom hur författarna vidareutvecklat Sydr med ändamålet att verktyget ska kunna upptäcka en bredare uppsättning sårbarheter.

Verktyget utvärderades i den ursprungliga publikationen utifrån hur korrekta genererade indata var, hur många kodgrenar som verktyget utforskat och hur lång tid analysen tog [39]. Under uppföljningsstudien genomförde författarna samma evaluering men de gjorde också tester på hur väl verktyget upptäckte sårbarheter i en testsvit [40]. Sårbarhetstesterna utvärderades på mängden sanna positiva, falska negativa och precision (andelen riktiga sårbarheter av de potentiella sårbarheterna).

Sydr finns tillgängligt på GitHub men endast som ett hybridfuzzer-verktyg[27]. Sydr som självständigt verktyg finns inte tillgängligt.

[23] https://github.com/angr/angr
[24] https://github.com/Inception-framework
[25] https://triton-library.github.io/
[26] https://dynamorio.org/
[27] https://sydr-fuzz.github.io/

4.3.3 Statiska analysverktyg

Trots att renodlade statiska tekniker var representerade i litteraturgranskningen fanns få verktyg som bedömdes ha en hög mognadsgrad. Endast ett verktyg i denna kategori nådde upp till TRL-nivå 8.

Goshawk är ett verktyg som inriktar sig på att hitta minneskorruptionsbuggar i källkod och presenteras i en publikation av Lyu m.fl. från 2022 [41]. Författarna trycker på att andra verktyg inte kan hitta buggar som existerar i specialskrivna minneshanteringsfunktioner utan kan endast hitta buggar där koden använder standardfunktioner såsom malloc och free. Verktyget använder sig av språkteknik (eng. natural language processing) och dataflödesanalys för att identifiera specialskrivna minneshanteringsfunktioner. När funktionerna har identifierats kan de analyseras genom att leta efter minneskorruptionsbuggar med en anpassad version av analysverktyget Clang Static Analyzer (CSA).[28] Författarna rapporterar att Goshawk har hittat 92 nya buggar.

Goshawk uppdaterades senast på GitHub i slutet av 2023.[29] Verktyget har utvecklats efter publiceringen, men större delen av utvecklingen genomfördes under 2021 och 2022.

4.3.4 Hur utvärderas verktygen?

I varje publikation förekommer det mer än ett sätt att påvisa ett verktygs effektivitet, exempelvis:

- kodtäckning
- tidsåtgång
- antal upptäckta sårbarheter i testsviter
- antal upptäckta buggar i produktionsmjukvara
- antal nya CVE:er som upptäckts i produktionsmjukvara.

Det finns ingen tydlig samsyn mellan de olika artiklarna om hur verktyg eller verktygstyper ska utvärderas eller jämföras. Detta har också påpekats i andra studier om kodanalysverktyg, exempelvis för fuzzers [42]. Vilka testsviter som används varierar också, till viss del beroende på plattform och språk som ska analyseras. Exempelvis behöver analysverktyg som inriktar sig på inbyggd mjukvara kunna hantera interaktionen med periferienheter och annan hårdvara, vilket ställer särskilda krav på verktygen. Ett exempel på ett sådant verktyg är Inception [37].

Verktygen med högst mognadsgrad utvärderades i respektive publikation enligt följande. Kompletterande information om respektive verktygs utvärdering finns i tabell 4.3.

[28] https://clang-analyzer.llvm.org/
[29] https://github.com/Yunlongs/Goshawk

AFLSmart utvärderades på både analysförmåga och hur effektivt fuzzern använder sin analystid. Verktygets analyskapacitet testades på en uppsättning produktionsmjukvara. Utöver detta mättes verktygets prestanda med antal buggar identifierade, krascher identifierade och kodtäckning [29].

Angr -plattformens sårbarhetsanalystekniker utvärderades med kodtäckning, mängd identifierade krascher och mängd falska positiva [36].

Fuzzware utvärderades på hur många sårbarheter som identifierades och hur mycket kodtäckning som verktyget kunde uppnå [34]. Kodtäckningen var dock det enda mätvärde som jämfördes med andra verktyg.

Goshawk utvärderades utifrån dess förmåga att hitta minneskorruptionsbuggar och hur bra den är på att identifiera minneshanteringsfunktioner [41]. Verktygets buggdetektion jämfördes med MallocChecker och K-MELD, samt SinkFinder för identifiering av minneshanteringsfunktioner.

Inception utvärderades utifrån fyra mätvärden: identifierade sårbarheter, rapporterade sårbarheter som var falska positiva, beräkningskostnad och analystid [37]. Mätvärdena undersöktes i två olika kontexter: i Klocwork där verktygets förmåga att upptäcka minneskorruptionssårbarheter i intermediärrepresentation undersöktes samt i produktionsmjukvara där sårbarheter från testsviten Juliet hade injicerats.

PolyFuzz utvärderades genom att testa verktyget på 15 flerspråkiga program respektive 15 enspråkiga program och mäta antal identifierade buggar och kodtäckning [35].

Sydr utvärderades utifrån följande mätvärden: generad indatas korrekthet, kodtäckning, analyshastighet, mängd sanna positiva, mängd falska negativa och verktygets precision [39, 40].

Sammanfattningsvis finns det två mätvärden som nästan alltid nyttjas för utvärdering av verktygen: antalet sårbarheter som identifierats samt kodtäckning. Dock finns det skillnader mellan hur mätvärdena benämns och vilka tillvägagångssätt som används för att utföra mätningarna.

Mätvärdet för antalet sårbarheter benämns oftast med antal krascher, buggar eller sårbarheter identifierade. Verktygen AFLSmart, Fuzzware och PolyFuzz går steget längre och presenterar även hur många av de identifierade sårbarheterna som blivit tilldelade CVE-nummer. Andra verktyg gör en djupare analys som undersöker hur många av de identifierade sårbarheterna är sanna respektive falska. Vidare genomfördes denna extra analys oftast av verktyg nyttjade testsviter för sin utvärdering, till exempel Angr [36], Inception [37] och Sydr [39, 40].

Även mätvärdet för kodtäckning gavs olika beteckningar och togs fram med olika tillvägagångsätt. Vanligt förekommande var att mäta mängden vägar och noder som verktygen sökt igenom i målmjukvaran.

Tabell 4.3. Kompletterande information om verktyg på TRL-nivå 8 och 9.

Verktyg	Jämförda verktyg	Testsviter/Dataset
AFLSmart	AFL, AFLFast, Peach, Vuzzer.	Egen uppsättning produktionsmjukvara.
Angr	-	DARPA CGC binärer.[30]
Fuzzware	µEMU,[31] P2IM[32]	Egen uppsättning produktionsmjukvara.
Goshawk	MallocChecker, K-MELD, SinkFinder.	Egen uppsättning produktionsmjukvara.
Inception	FIE, SURROGATES, Avatar.	Juliet C/C++ 1.3, [33] Klocwork Test Suite for C/C++.[34]
PolyFuzz	Hongfuzz,[35] Jazzer,[36] Atheris, [37] AFL++.[38]	Egen uppsättning produktionsmjukvara.
Sydr	-	Juliet C/C++ Dynamic Test Suite.[39]

[30] https://github.com/CyberGrandChallenge/samples
[31] https://github.com/MCUSec/uEmu
[32] https://github.com/RiS3-Lab/p2im
[33] https://samate.nist.gov/SARD/test-suites/112
[34] https://samate.nist.gov/SARD/test-suites/106
[35] https://github.com/google/honggfuzz
[36] https://github.com/CodeIntelligenceTesting/jazzer
[37] https://github.com/google/atheris
[38] https://github.com/AFLplusplus/AFLplusplus
[39] https://github.com/ispras/juliet-dynamic

5 Diskussion

I praktiskt arbete behövs verktyg som pålitligt kan upptäcka en stor mängd sårbarheter och som samtidigt är enkla att använda. Det är inte krav som uppfylls av enskilda akademiskt utvecklade verktyg. Att utveckla en produkt som omhändertar en stor mängd olika typer av sårbarheter är ett komplext åtagande. Det verkar sannolikt att det krävs ett ramverk som integrerar många olika tekniker för sårbarhetsupptäckt för att nå en bred täckning.

Detta kapitel diskuterar praktiska användningsaspekter som bidrar till en högre mognadsgrad för verktyg. Det diskuterar också studiens avgränsning till verktyg hämtade ur forskningslitteraturen.

5.1 Praktisk användning

För att ett mjukvaruanalysverktyg ska få praktisk användning i utvecklingsarbetet behöver det vara tillräckligt bra, oavsett om det utvecklas kommersiellt eller ideellt. Hur bra ett verktyg är kan utvärderas ur många perspektiv och på en mer eller mindre teknisk nivå. Detta avsnitt fokuserar på tre faktorer:

- Applicerbarhet – Om verktyget kan analysera mjukvaran som utvecklas.
- Effektivitet – Om verktyget kan identifiera sårbarheter som utvecklare vill undvika.
- Användbarhet – Om verktyget är enkelt nog att använda.

Om någon av dessa faktorer inte uppfylls tillräckligt väl finns risken att verktyget inte används.

5.1.1 Applicerbarhet

Artiklarna som fulltextgranskades har ett tydligt fokus på verktyg för C och C++, vilket också har påpekats i andra studier, exempelvis i en studie om statiska analysverktyg av Amankwah m. fl. [43]. Att fokus ligger på sårbarheter i C och C++ avspeglas även i datamängderna som används för att träna kodanalysverktyg, vilket visas i en annan av projektets studier av Jensen m.fl. [44]. En anledning kan vara att sårbarheter i programmeringsspråk som inte är minnessäkra, såsom buffertöverskridningar i C och C++, är relativt enkla att upptäcka. I mer komplexa programkonstruktioner blir det däremot svårare att upptäcka även sådana sårbarheter. Detta gäller speciellt i de fall där bristerna består av flera samverkande komponenter som är utspridda i mjukvaran.

Det är tydligt att vissa verktyg inte kan identifiera sårbarheter när mjukvaran blir mer komplex. En försvårande faktor är att komplexa och stora mjukvaror medför att det inte alltid räcker att analysera den egna koden utan även extern mjukvara måste analyseras, såsom dynamiska bibliotek som laddas in och öppen källkod som

kompileras in [45]. Även om publikationerna för de mest mogna verktygen diskuterar problemen med effektivitet, skalbarhet och komplexitet [29, 34, 41] handlar diskussionerna oftast om specifika sårbarheter i specifika kontexter.

Inget verktyg är applicerbart på alla typer av mjukvaror och system. Exempelvis påpekar Chen m.fl. att fuzzning i allmänhet verkar prestera sämre för inbyggda system, bland annat på grund av den begränsade gränsytan till systemet och på grund av bristande felhanteringsmekanismer [19]. Författarna till Fuzzware beskriver att fuzzning för inbyggda system är problematiskt och valde därför att lyfta ut mjukvaran till en emulator [34]. Även verktyget Inception lyfter ut mjukvaran till en emulator för att utföra tester på inbyggda system [37].

I en studie av Sutter m.fl. [16] analyserades 43 verktyg för att analysera sårbarheter i Android-mjukvara. Studien visade att de vanligaste teknikerna som användes för att hitta sårbarheter i Android-mjukvara är att spåra och analysera systemanrop samt att injicera kod i körande programprocesser (kallat dynamisk binär instrumentering). Fuzzning var inte ett givet val för analys av Android-mjukvara.

Befintlig forskning bygger ofta vidare på verktyg som tagits fram av andra forskningsgrupper. Alla verktyg med hög mognadsgrad bygger vidare på andra programvaror utom möjligtvis angr. AFLSmart och Fuzzware använder AFL som grund, medan PolyFuzz använder, en utveckling av AFL, AFL++ som grund. Inception utökar KLEE för sin analys, medan Sydr utökar verktygen Triton och DynamoRIO med en förbättrad symbolisk analys. Goshawk använder Clang Static Analyzer för analys. Det finns behov av att bredda perspektivet i forskningen för att täcka in fler typer av sårbarheter, programkonstruktioner, programmeringsspråk och miljöer.

Sammantaget är forskningen om kodanalysverktyg väldigt bred samtidigt som den är väldigt smal. Problemrymden utgörs av kombinationer av metoder, programmeringsspråk, programkonstruktioner, plattformar och sårbarheter. Sett till mängden kombinationer som kan utforskas är det endast en liten andel som har belysts. Det behövs därför fortsatt forskning inom området.

5.1.2 Effektivitet

Det är osannolikt att enskilda verktyg kommer att kunna täcka in alla aspekter av mjukvarusäkerhet. Amerikanska NSA har publicerat ett utlåtande om kodanalysverktygs begränsade effekt för minnessäkerhet i minnesosäkra programmeringsspråk [32]:

> *Varken statiska eller dynamiska kodanalysverktyg kan göra minnesosäker kod helt minnessäker. Eftersom alla verktyg har styrkor och svagheter rekommenderas det att använda flera verktyg för att öka chanserna att minnessårbarheter eller andra problem identifieras.*[40]

Det är uppenbart att olika ansatser till kodanalys hittar olika mängder av sårbarheter, vilket också ses i jämförelserna mellan verktygen där författarna ofta visar på att

[40] Författarnas översättning.

deras verktyg hittat fler sårbarheter än andra verktyg och identifierat ett antal nya CVE:er [29, 34, 35].

Tillsammans med myndigheter i USA, Australien, Kanada, Nya Zeeland och Storbritannien har amerikanska cybersäkerhetsmyndigheten CISA under de senaste åren publicerat information om de mest utnyttjade sårbarheterna [46]. Vid jämförelse mellan denna rapports studie finns det ett överlapp mellan CWE:erna som utnyttjas och CWE:erna verktygen kan identifiera. Detta gäller exempelvis CWE-20 och CWE-787. Däremot finns det även en stor mängd utnyttjade CWE:er som verktygen inte identifierar. Det kan inte heller förväntas givet den omfattning och bredd som sårbarheterna innebär. En av rekommendationerna i skrivelsen är att kodanalys bör nyttjas av utvecklingsorganisationer [46]:

> *Använd statisk och dynamiska säkerhetstestningsverktyg för att analysera källkod och applikationsbeteenden för att upptäcka felbenägna arbetssätt.*[41]

Tillvägagångssätten för att utvärdera och jämföra verktyg är inte standardiserade. Enbart sett till artiklarna som presenterar verktygen med högst mognadsgrad används åtskilliga mått. Inget av måtten är orimligt att använda sig av men för att underlätta för mjukvaruutvecklare att välja bland verktyg behöver det utvecklas ett standardiserat sätt att jämföra. I nuläget finns flera olika datamängder med testfall för att utvärdera kodanalysverktyg, exempelvis testsviten Juliet som användes vid utvärdering av flera verktyg med högst mognadsgrad. Datamängder med sårbar kod är dock inte heller utan kritik och behöver också utvärderas och utvecklas [44].

5.1.3 Användbarhet

Forskning om tekniker för kodanalys är typiskt sett inriktad på tekniska frågeställningar med djupgående analyser om mjukvara och dess exekvering. Vissa publikationer om kodanalysverktyg belyser också användbarhet, exempelvis Goshawk av Lyu m. fl. [41], vilket kan anses vara en viktig del i att omsätta en teknik till ett praktiskt användbart verktyg. Karlzén m. fl. identifierade flera problemområden som relaterar till användbarheten hos kodanalysverktyg [7]:

- Erfarenhet är viktig för att ha nytta av statiska säkerhetsanalysverktyg.
- Verktyg för att utveckla och testa mjukvara saknar flera saker ur ett användbarhetsperspektiv.
- Verktyg för att utveckla och testa mjukvara behöver en avvägning mellan att producera snabba varningar om sårbarheter och arbetsro för utvecklarna.
- Verktyg för statisk säkerhetstestning är generellt sett av begränsad nytta. Inget sådant verktyg kan hitta sårbarheter av alla typer och olika verktyg är bra på att hitta olika typer
- Verktyg för formella tekniker kan stödja sårbarhetsupptäckt, men ställer krav på att mjukvaran anpassas efter det.

[41] Författarnas översättning.

Vissa problem är svåra att komma ifrån, såsom att erfarenhet är viktigt för att kunna identifiera falska respektive sanna positiva bland felen som verktyg rapporterar. Även där finns förbättringspotential hos verktygen i termer av ökad precision i rapporterade sårbarheter för att minska bedömningsbehovet. Nachtigall m.fl. utförde en studie om hur kodanalysverktyg interagerar med slutanvändare [47]. I den identifierades en mängd brister hos ett stort antal statiska kodanalysverktyg. Ett grundläggande problem som identifierats i mer än en artikel är att verktygen inte alltid är möjliga att installera eller exekvera, exempelvis på grund av dålig dokumentation eller svårmötta beroenden [12, 47]. Flera andra studier har också anmärkt på användbarhet hos verktyg i termer av interaktionen med utvecklare [48] eller hur väldokumenterade verktygen är [49].

En äldre studie av Cohen m.fl. visade på att white-box-fuzzing-verktyget IntelliTest tillsammans med programmeringsparadigmet *design by contract* upplevdes som positivt av de som ingick i studien [14]. Design by contract är en metod där varje del av ett program har tydliga regler för vad som måste gälla innan och efter att den används, för att förhindra fel. I en annan studie om verktyg för formella tekniker uttrycker Bildsten m.fl. en liknande slutsats [21]:

> *Om verktygsstödet tas in under programmeringsfasen kan källkoden utformas och anpassas så att verktygen blir lätta att använda.*

Även om båda studierna får anses vara begränsade tyder de på att utvecklares arbetssätt kan behöva anpassas för att fullt ut dra nytta av verktygen.

5.2 Verktyg som inte återfinns i forskningslitteraturen

Under studiens gång har en stor mängd vetenskapliga publikationer granskats och kategoriserats, men med ett relativt litet utfall i verktyg som bedömts ha en hög mognadsnivå. Studien har fokuserat på akademiska publikationer från de senaste tio åren och medvetet avgränsat bort kommersiella och verktyg med öppen källkod som inte redan varit en del av de identifierade underlagen. Däremot går det inte att bortse från att kommersiella och öppna kodanalysverktyg har en stor roll inom mjukvaruutveckling.

En begränsning i denna litteraturstudie är alltså avgränsningen till att bara undersöka verktyg och tekniker som beskrivs i akademisk litteratur. Det finns många verktyg för mjukvaruanalyser som tillhandahålls kommersiellt eller som öppen källkod, men genom avgränsningen har dessa inte undersökts i studien. En anledning till avgränsningen är att kommersiellt utvecklade verktyg i allmänhet inte har samma nivå av transparens som akademiskt utvecklade verktyg. Det kan innebära svårigheter att få information om vilka tekniker verktygen bygger på, hur dessa appliceras och vilken effektivitet verktygen har. Verktyg med öppen källkod kan gå att undersöka, men dokumentationen är ofta begränsad. För kommersiella verktyg går det i regel inte att ta reda på exakt hur de fungerar. Trots detta utvärderar akademisk litteratur både verktyg med öppen källkod (se exempelvis [21]) och kommersiella verktyg (se exempelvis [50]). I en tidigare FOI-studie utvärderades bland annat det kommersiella verktyget CodeSonar [21]:

> *En nackdel med CodeSonar i denna rapports kontext är bristen på information om de metoder som verktyget använder internt. Det vore önskvärt att få mer information om vilka principer som ligger bakom analyserna för att kunna utvärdera verktygets styrka och i vilken utsträckning som analyserna hittar de kategorier av programmeringsfel som söks.*

I praktiken går det inte att bortse från verktyg utanför forskningssfären då många av dem har en utbredd praktisk användning hos företag och andra organisationer. Exempel på välanvända kodanalysverktyg Valgrind, [42] SonarQube, [43] och CodeSonar[44] samt analysverktyg inbyggda i utvecklingsmiljöer såsom Visual Studio[45] och IntelliJ IDEA.[46]

En aspekt värd att notera är att verktyg med öppen källkod används som bas för att utveckla nya verktyg inom forskning. Exempelvis är AFL, CLang[47] och KLEE [37] vanligt förekommande och används i praktiken även i sina grundutföranden [29, 34, 37, 41]. Dessa har inte analyserats som en del av studien eftersom de antingen utvecklats innan 2014 eller inte utvecklats som akademiska kodanalysverktyg.

[42] https://valgrind.org/
[43] https://www.sonarsource.com/products/sonarqube/
[44] https://codesecure.com/our-products/codesonar/
[45] https://visualstudio.microsoft.com/
[46] https://www.jetbrains.com/idea/
[47] https://clang.org/

6 Slutsats

Forskning om tekniker och kodanalysverktyg för att identifiera mjukvarusårbarheter är omfattande. Få verktyg med hög mognadsnivå har dock publicerats i form av akademiska publikationer. Det ska inte tolkas som att verktyg som används i praktiken är omogna då det finns en mängd kodanalysverktyg som producerats utanför forskningsinitiativ eller som låg utanför avgränsningarna i studien.

Studiens tre forskningsfrågor besvaras nedan.

Vilka verktyg finns det som implementerar tekniker för att identifiera potentiella sårbarheter?

Studien har identifierat 237 verktyg där mognadsgraden har kunnat skattas. Skattningen har baserats på respektive publikations redogörelse, utan att verifieras genom exempelvis tester. Många identifierade verktyg är att betrakta som vidareutvecklingar av andra verktyg, där justeringar har gjorts för att bättre träffa en specifik sårbarhet.

Av de identifierade verktygen bedöms sju hålla en hög mognadsgrad (TRL 8–9), där de som lägst kan anses vara kompletta och bekräftade i en operationell miljö. Dessa verktyg är: AFLSmart, Angr, Fuzzware, Goshawk, Inception, PolyFuzz och Sydr. På en medelhög mognadsgrad (TRL 5–7) återfinns 174 verktyg där de som lägst validerats i en relevant miljö. I den nedre änden av mognadsskalan (TRL 1–4) återfinns 56 verktyg som bedöms som på sin höjd experimentellt bevisade koncept.

Alla identifierade verktyg finns beskrivna i bilaga B.

Vilka kategorier av sårbarheter (inkl. språk, sårbarhetstyp, m.m.) kan respektive verktyg identifiera?

Verktygen med högst mognadsgrad i studien fokuserar på ett begränsat antal sårbarheter. I huvudsak handlar det om sårbarheter som uppstår till följd av minnesrelaterade problem som främst förekommer i minnesosäkra språk såsom C och C++. Exempel på brister som kan identifieras är buffertöverskridning, användning av nullpekare, heap/stack-överskridning, användning av frigjort minne, överskridning av talområden samt division med noll. Forskningslitteraturen varierar stort i hur de beskriver vilka sårbarheter som analyseras och hur verktygen utvärderas, vilket gör det svårt att ge ett entydigt svar på forskningsfrågan.

De verktyg som identifierats riktar i allmänhet in sig på några få och ofta närbesläktade sårbarheter, vilket begränsar den effekt som verktygen kan få i praktiskt arbete. Denna begränsning innebär att verktygen ofta inte kan användas för att hitta oväntade eller ovanliga brister i mjukvaran.

Hur mogna är verktygen för att identifiera potentiella sårbarheter? Finns det färdiga verktyg som är praktiskt användbara i utvecklings- och produktionsmiljö?

Studien har visat att det finns få identifierade verktyg med akademiskt ursprung som nått så pass hög mognadsgrad att de kan anses vara lämpliga vid praktiskt utvecklingsarbete. De flesta av verktyg verkar nå medelhög mognad (motsvarande TRL 5–6) innan de överges av sina utvecklare. Dessa är därmed inte tillräckligt färdiga för att kunna anses vara användbara i mjukvaruutveckling.

Utöver de akademiska verktygen finns verktyg med öppen källkod och kommersiella verktyg. Bland dem finns flera verktyg som är såväl mogna som praktiskt användbara, men de har inte ingått i denna studie.

Forskning kring hybridtekniker, där flera olika tekniker samverkar, är en potentiell väg för att bygga verktyg som är bredare i sitt angreppssätt. Antalet artiklar som beskriver verktyg med hybridtekniker har ökat under perioden 2014–2024, men de är fortfarande ovanliga bland de verktyg som nått hög mognadsgrad.

Det är av intresse att fortsätta studera tekniker som ligger strax utanför vad rapporten bedömer som mogna tekniker. I gruppen av verktyg med medelhög mognadsgrad, det vill säga TRL-nivåerna 5–7, kan det finnas verktyg som givet mer arbete kan stiga i mognadsgrad och på sikt vara praktiskt användbara.

7 Referenser

[1] J. Ohlin. *Unik IT-attack fortsätter att växa*. `https://www.svt.se/nyheter/inrikes/unik-it-attack-fortsatter-att-vaxa` [Besökt: 2024-11-11]. 2017.

[2] S. Cosar. *Spionprogram användes mot journalister och politiker*. `https://www.svt.se/nyheter/utrikes/spionprogram-anvandes-mot-journalister` [Besökt: 2024-11-11]. 2021.

[3] CISA. *Cost of a Cyber Incident: Systematic Review and Cross-Validation*. 2020.

[4] Försvarsmakten. *Försvarsmaktens Stratetiska Inriktning 2021-2030*. 2021.

[5] Försvarsmakten. *Militärstrategisk doktrin – MSD 22*. 2022.

[6] CISA, NSA, FBI, ACSC, CCCS, CERTNZ, NCSC-NZ, NCSC-UK, BSI, NCSC-NL, NCSC-NO, NÚKIB, INCD, KISA, NISC-JP, JPCERT/CC, CSA och CSIRTAMERICAS. *Secure by Design*. 2024.

[7] H. Karlzén, D. Eidenskog, J. Falkcrona och C. Valassi. *Varför har mjukvaror sårbarheter?* FOI-R--5550--SE. Totalförsvarets forskningsinstitut, 2023.

[8] T. J. McCabe, D. R. Wallace och A. H. Watson. *Structured Testing: A Testing Methodology Using the Cyciomatic Complexity Metric*. NIST Special Publication 500-235. NIST, 1996.

[9] IEC61508-7. *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures*. International Standard. IEC, 2010.

[10] S. Acharya och V. Pandya. “Bridge between black box and white box–gray box testing technique”. I: *International Journal of Electronics and Computer Science Engineering* 2.1 (2012), s. 175–185.

[11] SS-EN 61508-3. *Funktionssäkerhet hos elektriska, elektroniska och programmerbara elektroniska säkerhetskritiska system – Del 3: Fordringar på programvara.* International Standard. IEC, 2011.

[12] A. Adhikari och P. Kulkarni. “Survey of techniques to detect common weaknesses in program binaries”. I: *Cyber Security and Applications* 3 (2025), s. 100061. https://doi.org/10.1016/j.csa.2024.100061.

[13] R. Amankwah, J. Chen, H. Song och P. Kudjo. “Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites”. I: *Software: Practice and Experience* (dec. 2022). https://doi.org/10.1002/spe.3181.

[14] M. Cohen, I. Rodhe och J. Löfvenberg. *White-box Fuzzing*. FOI-R--4329--SE. Totalförsvarets forskningsinstitut, 2016.

[15] O. Zaazaa och H. El Bakkali. “Dynamic vulnerability detection approaches and tools: State of the Art”. I: *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*. 2020, s. 1–6. https://doi.org/10.1109/ICDS50568.2020.9268686.

[16] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach och J. Klein. “Dynamic Security Analysis on Android: A Systematic Literature Review”. I: *IEEE Access* 12 (2024), s. 57261–57287. https://doi.org/10.1109/ACCESS.2024.3390612.

[17] T. Ball och J. Daniel. “Deconstructing Dynamic Symbolic Execution”. I: *Dependable Software Systems Engineering*. 2015. URL: `https://api.semanticscholar.org/CorpusID:15763279`.

[18] B. S. Pak. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution*. Carnegie Mellon University, 2012.

[19] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo och W. Liu. “A systematic review of fuzzing techniques”. I: *Computers & Security* 75 (2018), s. 118–137. https://doi.org/10.1016/j.cose.2018.02.002.

[20] J. Fang och H. Qu. “VeriOover: A Verifier for Detecting Integer Overflow by Loop Abstraction”. I: *13th International Workshop on Computer Science and Engineering, WCSE 2023* (2023). Cited by: 0; All Open Access, Bronze Open Access, s. 13–19. https://doi.org/10.18178/wcse.2023.06.003.

[21] C. Bildsten, M. Cohen, D. Eidenskog och I. Rodhe. *Praktisk mjukvaruverifiering med formella metoder*. FOI-R--4642--SE. Totalförsvarets forskningsinstitut, 2018.

[22] M. Kalech. “Cyber-attack detection in SCADA systems using temporal pattern recognition techniques”. I: *Computers & Security* 84 (2019), s. 225–238. https://doi.org/10.1016/j.cose.2019.03.007.

[23] G. Lin, S. Wen, Q.-L. Han, J. Zhang och Y. Xiang. “Software Vulnerability Detection Using Deep Neural Networks: A Survey”. I: *Proceedings of the IEEE* 108.10 (2020), s. 1825–1848. https://doi.org/10.1109/JPROC.2020.2993293.

[24] A. Zhang, Y. Zhang, Y. Xu, C. Wang och S. Li. “Machine Learning-Based Fuzz Testing Techniques: A Survey”. I: *IEEE Access* 12 (2024), s. 14437–14454. https://doi.org/10.1109/ACCESS.2023.3347652.

[25] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey och S. Linkman. “Systematic literature reviews in software engineering – A systematic literature review”. I: *Information and Software Technology* 51.1 (2009). Special Section - Most Cited Articles in 2002 and Regular Research Papers, s. 7–15. https://doi.org/10.1016/j.infsof.2008.09.009.

[26] M. Granåsen, P.-A. Oskarsson, M. Olsén och N. Hallberg. *Tvärsektoriell krishantering: Värdering av förmåga och modellering av system*. FOI-R--5022--SE. Totalförsvarets forskningsinstitut, 2021.

[27] J. C. Mankins. *Technology readiness levels: A white paper*. White Paper. NASA, 1995.

[28] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger och S. Chakraborty. “Automatic Identification of Vulnerable Code: Investigations with an AST-Based Neural Network”. I: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021, s. 1475–1482. https://doi.org/10.1109/COMPSAC51774.2021.00219.

[29] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu och A. Roychoudhury. “Smart Greybox Fuzzing”. I: *IEEE Transactions on Software Engineering* 47.9 (2021), s. 1980–1997. https://doi.org/10.1109/TSE.2019.2941681.

[30] S. Cass. *The Top Programming Languages 2024*. `https://spectrum.ieee.org/top-programming-languages-2024` [Besökt 2024-11-11]. 2024.

[31] D. Gewirtz. *The most popular programming languages in 2024 (and what that even means)*. `https://www.zdnet.com/article/the-most-popular-programming-languages-in-2024-and-what-that-even-means/` [Besökt: 2024-11-11]. 2024.

[32] NSA. *Software Memory Safety*. 2023.

[33] K. D. Cooper och L. Torczon. “Chapter 5 - Intermediate Representations”. I: *Engineering a Compiler (Second Edition)*. Morgan Kaufmann, 2012, s. 221–268. https://doi.org/https://doi.org/10.1016/B978-0-12-088478-0.00005-0.

[34] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz och A. Abbasi. “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing”. I: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, juni 2022, s. 1239–1256.

[35] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo och H. Cai. “PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems”. I: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, juni 2023, s. 1379–1396.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel och G. Vigna. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. I: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, s. 138–157. https://doi.org/10.1109/SP.2016.17.

[37] N. Corteggiani, G. Camurati och A. Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software”. I: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, aug. 2018, s. 309–326.

[38] *The LLVM Compiler Infrastructure*. `https://llvm.org/` [Besökt: 2024-11-27].

[39] A. Vishnyakov, V. Logunova, E. Kobrin, D. Kuts, D. Parygina och A. Fedotov. “Symbolic Security Predicates: Hunt Program Weaknesses”. I: *2021 Ivannikov Ispras Open Conference (ISPRAS)*. 2021, s. 76–85. https://doi.org/10.1109/ISPRAS53967.2021.00016.

[40] A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky och S. Kurmangaleev. “Sydr: Cutting Edge Dynamic Symbolic Execution”. I: *2020 Ivannikov Ispras Open Conference (ISPRAS)*. 2020, s. 46–54. https://doi.org/10.1109/ISPRAS51486.2020.00014.

[41] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu och J. Li. “Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis”. I: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, s. 2096–2113. https://doi.org/10.1109/SP46214.2022.9833613.

[42] G. Klees, A. Ruef, B. Cooper, S. Wei och M. Hicks. “Evaluating fuzz testing”. I: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, s. 2123–2138.

[43] R. Amankwah, J. Chen, H. Song och P. K. Kudjo. “Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites”. I: *Software: Practice and Experience* 53.5 (2023), s. 1125–1143.

[44] C. Jensen, C. Vestlund, C. Gustavsson, V. Andersson, L. Nyholm och D. Eidenskog. *Inventering av testfall för verktyg som identifierar mjukvarusårbarheter*. FOI Memo 8673. Totalförsvarets forskningsinstitut, 2024.

[45] Y. Ning, Y. Zhang, C. Ma, Z. Guo och L. Yu. “Empirical Study of Software Composition Analysis Tools for C/C++ Binary Programs”. I: *IEEE Access* 12 (2024), s. 50418–50430. https://doi.org/10.1109/ACCESS.2023.3341224.

[46] CISA, FBI, NSA, ACSC, CCCS, NCSC-NZ, C. NZ och NCSC-UK. *2023 Top Routinely Exploited Vulnerabilities*. 2024.

[47] M. Nachtigall, M. Schlichtig och E. Bodden. “A large-scale study of usability criteria addressed by static analysis tools”. I: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, s. 532–543.

[48] M. Tahaei, K. Vaniea, K. Beznosov och M. K. Wolters. “Security notifications in static analysis tools: Developers’ attitudes, comprehension, and ability to act on them”. I: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, s. 1–17.

[49] C. Latappy, T. Degueule, J.-R. Falleri, R. Robbes, X. Blanc och C. Teyton. “What the Fix? A Study of ASATs Rule Documentation”. I: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 2024, s. 246–257.

[50] V. Bhutani, F. G. Toosi och J. Buckley. “Analysing the Analysers: An Investigation of Source Code Analysis Tools”. I: *Applied Computer Systems* 29.1 (2024), s. 98–111.

[51] J. Akram och P. Luo. “SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment”. I: *Software: Practice and Experience* 51.2 (2021), s. 294–318. https://doi.org/10.1002/spe.2905.

[52] Z. Wu, K. Lu, X. Wang och X. Zhou. “Collaborative Technique for Concurrency Bug Detection”. I: *International Journal of Parallel Programming* 43.2 (febr. 2014), s. 260–285. https://doi.org/10.1007/s10766-014-0304-y.

[53] H. Liang, L. Wang, D. Wu och J. Xu. “MLSA: A static bugs analysis tool based on LLVM IR”. I: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 2016, s. 407–412. https://doi.org/10.1109/SNPD.2016.7515932.

[54] M. Ehrenberg, S. Sarkani och T. A. Mazzuchi. “Python source code vulnerability detection with named entity recognition”. I: *Computers & Security* 140 (2024), s. 103802. https://doi.org/10.1016/j.cose.2024.103802.

[55] M. Saletta och C. Ferretti. “A Neural Embedding for Source Code: Security Analysis and CWE Lists”. I: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 2020, s. 523–530. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00095.

[56] J. Nam, S. Wang, Y. Xi och L. Tan. “A bug finder refined by a large set of open-source projects”. I: *Information and Software Technology* 112 (2019), s. 164–175. https://doi.org/10.1016/j.infsof.2019.04.014.

[57] C. Do Xuan, D. H. Mai, M. C. Thanh och B. Van Cong. “A novel approach for software vulnerability detection based on intelligent cognitive computing”. I: *The Journal of Supercomputing* 79.15 (maj 2023), s. 17042–17078. https://doi.org/10.1007/s11227-023-05282-4.

[58] Y. Xiao, Z. Xu, W. Zhang, C. Yu, L. Liu, W. Zou, Z. Yuan, Y. Liu, A. Piao och W. Huo. “VIVA: Binary Level Vulnerability Identification via Partial Signature”. I: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2021, s. 213–224. https://doi.org/10.1109/SANER50967.2021.00028.

[59] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty och Y. Xiong. “Project Achilles: A Prototype Tool for Static Method-Level Vulnerability Detection of Java Source Code Using a Recurrent Neural Network”. I: *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2019, s. 114–121. https://doi.org/10.1109/ASEW.2019.00040.

[60] M. Shudrak. “WinHeap Explorer: Efficient and Transparent Heap-Based Bug Detection in Machine Code”. I: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, s. 94–101. https://doi.org/10.1109/QRS.2017.20.

[61] G. Stergiopoulos, P. Katsaros och D. Gritzalis. “Execution Path Classification for Vulnerability Analysis and Detection”. I: *E-Business and Telecommunications*. Springer International Publishing, 2016, s. 293–317. https://doi.org/10.1007/978-3-319-30222-5_14.

[62] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak och L. Karaçay. “Vulnerability Prediction From Source Code Using Machine Learning”. I: *IEEE Access* 8 (2020), s. 150672–150684. https://doi.org/10.1109/ACCESS.2020.3016774.

[63] J. Pewny, B. Garmany, R. Gawlik, C. Rossow och T. Holz. “Cross-Architecture Bug Search in Binary Executables”. I: *2015 IEEE Symposium on Security and Privacy*. 2015, s. 709–724. https://doi.org/10.1109/SP.2015.49.

[64] D. Yan, L. Pan, R. Yan, J. Yan och J. Zhang. “Comprehensive Static Analysis for Configurable Software via Combinatorial Instantiation”. I: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2017, s. 67–74. https://doi.org/10.1109/COMPSAC.2017.91.

[65] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood och M. McConley. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. I: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, s. 757–762. https://doi.org/10.1109/ICMLA.2018.00120.

[66] H. Yan, Y. Sui, S. Chen och J. Xue. “Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities”. I: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, s. 327–337. https://doi.org/10.1145/3180155.3180178.

[67] W. Jin, S. Ullah, D. Yoo och H. Oh. “NPDHunter: Efficient Null Pointer Dereference Vulnerability Detection in Binary”. I: *IEEE Access* 9 (2021), s. 90153–90169. https://doi.org/10.1109/ACCESS.2021.3091209.

[68] G. Lu, X. Ju, X. Chen, W. Pei och Z. Cai. “GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning”. I: *Journal of Systems and Software* 212 (2024), s. 112031. https://doi.org/10.1016/j.jss.2024.112031.

[69] W. Li, D. Xu, W. Wu, X. Gong, X. Xiang, Y. Wang, F. gu och Q. Zeng. “Memory access integrity: detecting fine-grained memory access errors in binary code”. I: *Cybersecurity* 2.1 (juni 2019). https://doi.org/10.1186/s42400-019-0035-x.

[70] B. Bowman och H. H. Huang. “VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets”. I: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2020, s. 53–69. https://doi.org/10.1109/EuroSP48549.2020.00012.

[71] C. Li, M. Zhou, Z. Gu, G. Chen, Y. Wang, J. Wu och M. Gu. “VBSAC: a value-based static analyzer for C”. I: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, s. 382–385. https://doi.org/10.1145/3293882.3338998.

[72] J. Gao, Y. Jiang, Z. Liu, X. Yang, C. Wang, X. Jiao, Z. Yang och J. Sun. “Semantic Learning and Emulation Based Cross-Platform Binary Vulnerability Seeker”. I: *IEEE Transactions on Software Engineering* 47.11 (2021), s. 2575–2589. https://doi.org/10.1109/TSE.2019.2956932.

[73] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu och A. Tiu. “Steelix: program-state based binary fuzzing”. I: ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, s. 627–637. https://doi.org/10.1145/3106237.3106295.

[74] B. Wang, K. Lu, Q. Wu och A. Pakki. “Unleashing Coveraged-Based Fuzzing Through Comprehensive, Efficient, and Faithful Exploitable-Bug Exposing”. I: *IEEE Transactions on Dependable and Secure Computing* 19.5 (2022), s. 2998–3010. https://doi.org/10.1109/TDSC.2021.3079857.

[75] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno och W. Lee. “VulChecker: graph-based vulnerability localization in source code”. I: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC ’23. Anaheim, CA, USA: USENIX Association, 2023.

[76] T. Wu, J. Liu, X. Deng, J. Yan och J. Zhang. “Relda2: an effective static analysis tool for resource leak detection in Android apps”. I: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE ’16. Singapore, Singapore: Association for Computing Machinery, 2016, s. 762–767. https://doi.org/10.1145/2970276.2970278.

[77] Y. Dong, Y. Tang, X. Cheng, Y. Yang och S. Wang. “SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding”. I: *Information and Software Technology* 158 (2023), s. 107168. https://doi.org/10.1016/j.infsof.2023.107168.

[78] A. Lee, I. Ariq, Y. Kim och M. Kim. “POWER: Program Option-Aware Fuzzer for High Bug Detection Ability”. I: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022, s. 220–231. https://doi.org/10.1109/ICST53961.2022.00032.

[79] F. K. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa och L. C. Cordeiro. “Combining BMC and Fuzzing Techniques for Finding Software Vulnerabilities in Concurrent Programs”. I: *IEEE Access* 10 (2022), s. 121365–121384. https://doi.org/10.1109/ACCESS.2022.3223359.

[80] X. Cheng, G. Zhang, H. Wang och Y. Sui. “Path-sensitive code embedding via contrastive learning for software vulnerability detection”. I: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, s. 519–531. https://doi.org/10.1145/3533767.3534371.

[81] P. Zeng, G. Lin, L. Pan, Y. Tai och J. Zhang. “Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: A Survey”. I: *IEEE Access* 8 (2020), s. 197158–197172. https://doi.org/10.1109/ACCESS.2020.3034766.

[82] J. Ma, P. Zhang, G. Dong, S. Shao och J. Zhang. “TWalker: An efficient taint analysis tool”. I: *2014 10th International Conference on Information Assurance and Security*. 2014, s. 18–22. https://doi.org/10.1109/ISIAS.2014.7064628.

[83] Z. Yang, Z. Yu, X. Su och P. Ma. “RaceTracker: Effective and efficient detection of data races”. I: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 2016, s. 293–300. https://doi.org/10.1109/SNPD.2016.7515908.

[84] J. Choi, J. Jang, C. Han och S. K. Cha. “Grey-Box Concolic Testing on Binary Code”. I: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, s. 736–747. https://doi.org/10.1109/ICSE.2019.00082.

[85] T. Liu, C. Curtsinger och E. D. Berger. “DoubleTake: fast and precise error detection via evidence-based dynamic analysis”. I: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, s. 911–922. https://doi.org/10.1145/2884781.2884784.

[86] Y. Wang, P. Jia, X. Peng, C. Huang och J. Liu. “BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention”. I: *Computers & Security* 125 (2023), s. 103023. https://doi.org/10.1016/j.cose.2022.103023.

[87] D. Hin, A. Kan, H. Chen och M. A. Babar. “LineVD: statement-level vulnerability detection using graph neural networks”. I: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, s. 596–607. https://doi.org/10.1145/3524842.3527949.

[88] Z. Li, J. Wang, M. Sun och J. C. Lui. “MirChecker: Detecting Bugs in Rust Programs via Static Analysis”. I: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, s. 2183–2196. https://doi.org/10.1145/3460120.3484541.

[89] B. Zhang, C. Feng, A. Herrera, V. Chipounov, G. Candea och C. Tang. “Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing”. I: *IET Software* 12.6 (2018), s. 507–519. https://doi.org/10.1049/iet-sen.2017.0200.

[90] Y. ZHUANG och Y.-N. TSENG. “A Novel Detection Method for the Security Vulnerability of Time-of-Check to Time-of-Use”. I: *Journal of Information Science and Engineering* 38.6 (nov. 2022), s. 1171–1188. https://doi.org/10.6688/JISE.202211_38(6).0005.

[91] R. Li, C. Zhang, C. Feng, X. Zhang och C. Tang. “Locating Vulnerability in Binaries Using Deep Neural Networks”. I: *IEEE Access* 7 (2019), s. 134660–134676. https://doi.org/10.1109/ACCESS.2019.2942043.

[92] A. Mahyari. “A Hierarchical Deep Neural Network for Detecting Lines of Codes with Vulnerabilities”. I: *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. 2022, s. 1–7. https://doi.org/10.1109/QRS-C57518.2022.00011.

[93] H. Zhang, A. Zhou, P. Jia, L. Liu, J. Ma och L. Liu. “InsFuzz: Fuzzing Binaries With Location Sensitivity”. I: *IEEE Access* 7 (2019), s. 22434–22444. https://doi.org/10.1109/ACCESS.2019.2894178.

[94] J. Park, J. Shin och B. Choi. “Detection of Vulnerabilities by Incorrect Use of Variable Using Machine Learning”. I: *Electronics* 12.5 (2023). https://doi.org/10.3390/electronics12051197.

[95] H. Li, J. Oh, H. Oh och H. Lee. “Automated Source Code Instrumentation for Verifying Potential Vulnerabilities”. I: *ICT Systems Security and Privacy Protection*. Utg. av J.-H. Hoepman och S. Katzenbeisser. Cham: Springer International Publishing, 2016, s. 211–226.

[96] D. Zhang, H. Xian, J. Chen och Z. Xu. “VDCNet: A Vulnerability Detection and Classification System in Cross-Project Scenarios”. I: *Artificial Neural Networks and Machine Learning – ICANN 2023*. Cham: Springer Nature Switzerland, 2023, s. 305–316.

[97] S. Godboley., K. Gupta. och R. G. Monika. “AV-AFL: A Vulnerability Detection Fuzzing Approach by Proving Non-reachable Vulnerabilities using Sound Static Analyser”. I: *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC. SciTePress, 2022, s. 301–308. https://doi.org/10.5220/0011032900003176.

[98] K. Zhu, Y. Lu och H. Huang. “Scalable Static Detection of Use-After-Free Vulnerabilities in Binary Code”. I: *IEEE Access* 8 (2020), s. 78713–78725. https://doi.org/10.1109/ACCESS.2020.2990197.

[99] F. Rustamov, J. Kim, J. Yu, H. Kim och J. Yun. “BugMiner: Mining the Hard-to-Reach Software Vulnerabilities through the Target-Oriented Hybrid Fuzzer”. I: *Electronics* 10.1 (2021). https://doi.org/10.3390/electronics10010062.

[100] T. Brennan, S. Saha, T. Bultan och C. S. Păsăreanu. “Symbolic path cost analysis for side-channel detection”. I: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, s. 27–37. https://doi.org/10.1145/3213846.3213867.

[101] J. D’Abruzzo Pereira och M. Vieira. “On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects”. I: *2020 16th European Dependable Computing Conference (EDCC)*. 2020, s. 97–102. https://doi.org/10.1109/EDCC51268.2020.00025.

[102] X. Cheng, H. Wang, J. Hua, G. Xu och Y. Sui. “DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network”. I: *ACM Trans. Softw. Eng. Methodol.* 30.3 (april 2021). https://doi.org/10.1145/3436877.

[103] Z. Li, J. Wang, M. Sun och J. C. S. Lui. “Detecting Cross-language Memory Management Issues in Rust”. I: *Computer Security – ESORICS 2022*. Springer Nature Switzerland, 2022, s. 680–700. https://doi.org/10.1007/978-3-031-17143-7_33.

[104] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Krügel och G. Vigna. “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware”. I: *Network and Distributed System Security Symposium*. 2015. URL: `https://api.semanticscholar.org/CorpusID:17298209`.

[105] C. Shao, G. Li, J. Wu och X. Zheng. “Exploring Semantic Redundancy using Backdoor Triggers: A Complementary Insight into the Challenges Facing DNN-based Software Vulnerability Detection”. I: 33.4 (april 2024). https://doi.org/10.1145/3640333.

[106] Z. Xu, J. Zhang och Z. Xu. “Melton: a practical and precise memory leak detection tool for C programs”. I: *Frontiers of Computer Science* 9.1 (dec. 2014), s. 34–54. https://doi.org/10.1007/s11704-014-3460-8.

[107] M. Wang, C. Tao och H. Guo. “LCVD: Loop-oriented code vulnerability detection via graph neural network”. I: *Journal of Systems and Software* 202 (2023), s. 111706. https://doi.org/10.1016/j.jss.2023.111706.

[108] S. Chen, Y. Zhang, L. Fan, J. Li och Y. Liu. “AUSERA: Automated Security Vulnerability Detection for Android Apps”. I: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. https://doi.org/10.1145/3551349.3559524.

[109] O. Hany och M. Abu-Elkheir. “Detecting Vulnerabilities in Source Code Using Machine Learning”. I: *Proceedings of the International Conference on Applied CyberSecurity (ACS) 2021*. Springer International Publishing, 2022, s. 35–41. https://doi.org/10.1007/978-3-030-95918-0_4.

[110] G. Zhang, P.-F. Wang, T. Yue, X.-D. Kong, X. Zhou och K. Lu. “ovAFLow: Detecting Memory Corruption Bugs with Fuzzing-Based Taint Inference”. I: *Journal of Computer Science and Technology* 37.2 (mars 2022), s. 405–422. https://doi.org/10.1007/s11390-021-1600-9.

[111] H. Yan, Y. Sui, S. Chen och J. Xue. “Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection”. I: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC ’17. Orlando, FL, USA: Association for Computing Machinery, 2017, s. 42–54. https://doi.org/10.1145/3134600.3134620.

[112] S. Ma, M. Jiao, S. Zhang, W. Zhao och D. W. Wang. “Practical null pointer dereference detection via value-dependence analysis”. I: *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2015, s. 70–77. https://doi.org/10.1109/ISSREW.2015.7392049.

[113] K. Vorobyov, N. Kosmatov och J. Signoles. “Detection of Security Vulnerabilities in C Code Using Runtime Verification: An Experience Report”. I: *Tests and Proofs*. Springer International Publishing, 2018, s. 139–156. https://doi.org/10.1007/978-3-319-92994-1_8.

[114] M. Alqaradaghi och T. Kozsik. “Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code”. I: *IEEE Access* 12 (2024), s. 55824–55842. https://doi.org/10.1109/ACCESS.2024.3389955.

[115] J. Ye, B. Zhang, R. Li, C. Feng och C. Tang. “Program State Sensitive Parallel Fuzzing for Real World Software”. I: *IEEE Access* 7 (2019), s. 42557–42564. https://doi.org/10.1109/ACCESS.2019.2905744.

[116] S. E. Ponta, H. Plate och A. Sabetta. “Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software”. I: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, s. 449–460. https://doi.org/10.1109/ICSME.2018.00054.

[117] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi och J. Hu. “VulPecker: an automated vulnerability detection system based on code similarity analysis”. I: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: Association for Computing Machinery, 2016, s. 201–213. https://doi.org/10.1145/2991079.2991102.

[118] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu och H. Jin. “VulCNN: an image-inspired scalable vulnerability detection system”. I: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, s. 2365–2376. https://doi.org/10.1145/3510003.3510229.

[119] J. Schilling och T. Müller. “VANDALIR: Vulnerability Analyses Based on Datalog and LLVM-IR”. I: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2022, s. 96–115. https://doi.org/10.1007/978-3-031-09484-2_6.

[120] S. Kim, J. Choi, M. E. Ahmed, S. Nepal och H. Kim. “VulDeBERT: A Vulnerability Detection System Using BERT”. I: *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2022, s. 69–74. https://doi.org/10.1109/ISSREW55968.2022.00042.

[121] X. Ma, J. Yan, W. Wang, J. Yan, J. Zhang och Z. Qiu. “Detecting Memory-Related Bugs by Tracking Heap Memory Management of C++ Smart Pointers”. I: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, s. 880–891. https://doi.org/10.1109/ASE51524.2021.9678836.

[122] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu och D. (Tian. “PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications”. I: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, aug. 2021, s. 249–266. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/kim`.

[123] Y. Rong, C. Zhang, J. Liu och H. Chen. “Valkyrie: Improving fuzzing performance through deterministic techniques”. I: *Journal of Systems and Software* 209 (2024), s. 111886. https://doi.org/10.1016/j.jss.2023.111886.

[124] X. Bai, L. Xing, M. Zheng och F. Qu. “iDEA: Static Analysis on the Security of Apple Kernel Drivers”. I: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, s. 1185–1202. https://doi.org/10.1145/3372297.3423357.

[125] Z. Chen, Q. Zhang, J. Wu, J. Yan och J. Xue. “A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety”. I: *IEEE Transactions on Software Engineering* 49.4 (2023), s. 2107–2127. https://doi.org/10.1109/TSE.2022.3210580.

[126] S. Zhu, J. Wang, J. Sun, J. Yang, X. Lin, T. Wang, L. Zhang och P. Cheng. “Better Pay Attention Whilst Fuzzing”. I: *IEEE Transactions on Software Engineering* 50.2 (2024), s. 190–208. https://doi.org/10.1109/TSE.2023.3338129.

[127] V. Sachidananda, S. Bhairav och Y. Elovici. “OVER: overhauling vulnerability detection for iot through an adaptable and automated static analysis framework”. I: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC ’20. Brno, Czech Republic: Association for Computing Machinery, 2020, s. 729–738. https://doi.org/10.1145/3341105.3373930.

[128] A. V. Rhein, J. Liebig, A. Janker, C. Kästner och S. Apel. “Variability-Aware Static Analysis at Scale: An Empirical Study”. I: *ACM Trans. Softw. Eng. Methodol.* 27.4 (nov. 2018). https://doi.org/10.1145/3280986.

[129] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu och L. Sun. “FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware”. I: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS ’23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, s. 138–152. https://doi.org/10.1145/3623278.3624759.

[130] U. Kargén och N. Shahmehri. “Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing”. I: ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, s. 782–792. https://doi.org/10.1145/2786805.2786844.

[131] Z. Fang, Q. Liu, Y. Zhang, K. Wang, Z. Wang och Q. Wu. “A static technique for detecting input validation vulnerabilities in Android apps”. I: *Science China Information Sciences* 60.5 (sept. 2016). https://doi.org/10.1007/s11432-015-5422-7.

[132] Y. Noller och S. Tizpaz-Niari. “QFuzz: quantitative fuzzing for side channels”. I: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, s. 257–269. https://doi.org/10.1145/3460319.3464817.

[133] H. Chen, Y. Li, B. Chen, Y. Xue och Y. Liu. “FOT: a versatile, configurable, extensible fuzzing framework”. I: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, s. 867–870. https://doi.org/10.1145/3236024.3264593.

[134] I. Baheux, O.-E.-K. Aktouf, M. E. A. Tebib, M. Graa, P. Andre och Y. Ledru. “DroidSecTester: Towards context-driven modelling and detection of Android application vulnerabilities”. I: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2023, s. 136–141. https://doi.org/10.1109/ISSREW60843.2023.00063.

[135] Y. David, N. Partush och E. Yahav. “FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware”. I: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. ACM, mars 2018. https://doi.org/10.1145/3173162.3177157.

[136] J. Chen, C. Zhang, S. Cai, L. Zhang och L. Ma. “A memory-related vulnerability detection approach based on vulnerability model with Petri Net”. I: *Journal of Logical and Algebraic Methods in Programming* 132 (april 2023), s. 100859. https://doi.org/10.1016/j.jlamp.2023.100859.

[137] P. Srivastava, F. Toffalini, K. Vorobyov, F. Gauthier, A. Bianchi och M. Payer. “Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities”. I: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’23. ACM, nov. 2023, s. 1586–1597. https://doi.org/10.1145/3611643.3616313.

[138] L. Sun, X. Li, H. Qu och X. Zhang. “AFLTurbo: Speed up Path Discovery for Greybox Fuzzing”. I: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, okt. 2020, s. 81–91. https://doi.org/10.1109/issre5003.2020.00017.

[139] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang, C. Hauser och Y. Shoshitaishvili. “Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs”. I: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, aug. 2022, s. 413–430.

[140] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye och L. Sun. “Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis”. I: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’23. ACM, juli 2023. https://doi.org/10.1145/3597926.3598062.

[141] A. Yadavally, Y. Li, S. Wang och T. N. Nguyen. “A Learning-Based Approach to Static Program Slicing”. I: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (april 2024), s. 83–109. https://doi.org/10.1145/3649814.

[142] H. Liang, L. Jiang, L. Ai och J. Wei. “Sequence Directed Hybrid Fuzzing”. I: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, febr. 2020, s. 127–137. https://doi.org/10.1109/saner48275.2020.9054807.

[143] R. Padhye, C. Lemieux, K. Sen, M. Papadakis och Y. Le Traon. “Semantic fuzzing with zest”. I: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’19. ACM, juli 2019, s. 329–340. https://doi.org/10.1145/3293882.3330576.

[144] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik och T. Holz. “REDQUEEN: Fuzzing with Input-to-State Correspondence”. I: *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS 2019. Internet Society, 2019. https://doi.org/10.14722/ndss.2019.23371.

[145] G. Palavicini Jr, J. Bryan, E. Sheets, M. Kline och J. San Miguel. “Towards Firmware Analysis of Industrial Internet of Things (IIoT) - Applying Symbolic Analysis to IIoT Firmware Vetting”. I: *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*. SCITEPRESS - Science och Technology Publications, 2017, s. 470–477. https://doi.org/10.5220/0006393704700477.

[146] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu och B. Mao. “PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary”. I: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS ’19. ACM, juli 2019. https://doi.org/10.1145/3321705.3329828.

[147] K. Even-Mendoza, A. Sharma, A. F. Donaldson och C. Cadar. “GrayC: Greybox Fuzzing of Compilers and Analysers for C”. I: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’23. ACM, juli 2023, s. 1219–1231. https://doi.org/10.1145/3597926.3598130.

[148] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida och H. Bos. “VUzzer: Application-aware Evolutionary Fuzzing”. I: *Proceedings 2017 Network and Distributed System Security Symposium*. NDSS 2017. Internet Society, 2017. https://doi.org/10.14722/ndss.2017.23404.

[149] H. Liang, X. Yu, X. Cheng, J. Liu och J. Li. “Multiple Targets Directed Greybox Fuzzing”. I: *IEEE Transactions on Dependable and Secure Computing* 21.1 (jan. 2024), s. 325–339. https://doi.org/10.1109/tdsc.2023.3253120.

[150] Y. Wang, Z. Wu, Q. Wei och Q. Wang. “NeuFuzz: Efficient Fuzzing With Deep Neural Network”. I: *IEEE Access* 7 (2019), s. 36340–36352. https://doi.org/10.1109/access.2019.2903291.

[151] H. Peng, Y. Shoshitaishvili och M. Payer. “T-Fuzz: Fuzzing by Program Transformation”. I: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2018, s. 697–710. https://doi.org/10.1109/sp.2018.00056.

[152] J. Pan, G. Yan och X. Fan. “Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities”. I: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, aug. 2017, s. 149–165. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/pan`.

[153] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda och M. Egele. “HotFuzz: Discovering Temporal and Spatial Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing”. I: *ACM Transactions on Privacy and Security* 25.4 (juli 2022), s. 1–35. https://doi.org/10.1145/3532184.

[154] J. Li, S. Li, K. Li, F. Luo, H. Yu, S. Li och X. Li. “ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems”. I: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. ACM, febr. 2024, s. 1–12. https://doi.org/10.1145/3597503.3623315.

[155] T. Yang, S. Lian, Q. Jia, C. Hu och S. Guo. “Multi-granularity Deep Vulnerability Detection Using Graph Neural Networks”. I: *Neural Information Processing*. Springer Nature Singapore, nov. 2023, s. 152–164. https://doi.org/10.1007/978-981-99-8184-7_12.

[156] W. Li, H. Yang, X. Luo, L. Cheng och H. Cai. “PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing”. I: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’23. ACM, nov. 2023, s. 1645–1659. https://doi.org/10.1145/3576915.3623166.

[157] Q. Zhao, C. Huang och L. Dai. “VULDEFF: Vulnerability detection method based on function fingerprints and code differences”. I: *Knowledge-Based Systems* 260 (jan. 2023), s. 110139. https://doi.org/10.1016/j.knosys.2022.110139.

[158] Q. Zhong, T. Peng och S. Chen. “VulDeCRG: A Vulnerability Detection System based on Code Representation Graph”. I: *2023 2nd International Conference on Artificial Intelligence, Human-Computer Interaction and Robotics (AIHCIR)*. IEEE, dec. 2023, s. 243–247. https://doi.org/10.1109/aihcir61661.2023.00048.

[159] M. T. Sultan och H. El Sayed. “A Deep Learning-based Approach for an Automated Vulnerability Detection in Source Code”. I: *2023 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT)*. IEEE, dec. 2023, s. 17–22. https://doi.org/10.1109/gcaiot61060.2023.10385129.

[160] M. Yoda, S. Nakamura, Y. Sei, Y. Tahara och A. Ohsuga. “A Middleware to Improve Analysis Coverage in IoT Vulnerability Detection”. I: *2023 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS)*. IEEE, nov. 2023, s. 103–107. https://doi.org/10.1109/iotais60147.2023.10346047.

[161] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng och C. Zhang. “KextFuzz: A Practical Fuzzer for macOS Kernel EXTensions on Apple Silicon”. I: *IEEE Transactions on Dependable and Secure Computing* 21.4 (juli 2024), s. 3453–3468. https://doi.org/10.1109/tdsc.2023.3330852.

[162] J. Fang och H. Qu. “VeriOover: A Verifier for Detecting Integer Overflow by Loop Abstraction”. I: *Proceedings of 2023 the 13th International Workshop on Computer Science and Engineering*. WCSE 2023. WCSE, 2023. https://doi.org/10.18178/wcse.2023.06.003.

[163] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel och T. Holz. “HOEDUR: embedded firmware fuzzing using multi-stream inputs”. I: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC ’23. Anaheim, CA, USA: USENIX Association, 2023.

[164] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller och T. Holz. “Fuzztruction: using fault injection-based fuzzing to leverage implicit domain knowledge”. I: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC ’23. Anaheim, CA, USA: USENIX Association, 2023.

[165] R. Li, B. Zhang, T. Wang och C. Tang. “A Heap Manipulation Diversity Fuzzing Method for Spatial Heap Vulnerabilities Exploitation”. I: *2023 8th International Conference on Intelligent Computing and Signal Processing (ICSP)*. IEEE, april 2023, s. 200–204. https://doi.org/10.1109/icsp58490.2023.10248640.

[166] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf och A.-R. Sadeghi. “DARWIN: Survival of the Fittest Fuzzing Mutators”. I: *Proceedings 2023 Network and Distributed System Security Symposium*. NDSS 2023. Internet Society, 2023. https://doi.org/10.14722/ndss.2023.23159.

[167] L. Chen, Y. Wang, J. Linghu, Q. Hou, Q. Cai, S. Guo och Z. Xue. “SaTC: Shared-Keyword Aware Taint Checking for Detecting Bugs in Embedded Systems”. I: *IEEE Transactions on Dependable and Secure Computing* 21.4 (juli 2024), s. 2421–2433. https://doi.org/10.1109/tdsc.2023.3307430.

[168] N. Cui, L. Chen och G. Shi. “Binary Code Vulnerability Location Identification with Fine-grained Slicing”. I: *2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. IEEE, febr. 2023, s. 502–506. https://doi.org/10.1109/acctcs58815.2023.00103.

[169] J.-J. Bai, Z.-X. Fu, K.-T. Xie och Z.-M. Jiang. “Testing Error Handling Code With Software Fault Injection and Error-Coverage-Guided Fuzzing”. I: *IEEE Transactions on Dependable and Secure Computing* 21.4 (juli 2024), s. 1724–1739. https://doi.org/10.1109/tdsc.2023.3288876.

[170] Z. Wang, S. Meng och Y. Chen. “Vulnerability Detection with Representation Learning”. I: *Ubiquitous Security*. Springer Nature Singapore, 2023, s. 116–128. https://doi.org/10.1007/978-981-99-0272-9_8.

[171] S. Chen, J. Nan, Z. Wu och Z. Wang. “JSVulExplorer: a JavaScript vulnerability detection model based on transfer learning”. I: *Fifth International Conference on Computer Information Science and Artificial Intelligence (CISAI 2022)*. Utg. av Y. Zhong. SPIE, mars 2023. https://doi.org/10.1117/12.2667324.

[172] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu och K. Lu. “VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search”. I: *Proceedings 2023 Network and Distributed System Security Symposium*. NDSS 2023. Internet Society, 2023. https://doi.org/10.14722/ndss.2023.24415.

[173] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang och Z. Xue. “SFuzz: Slice-based Fuzzing for Real-Time Operating Systems”. I: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. ACM, nov. 2022, s. 485–498. https://doi.org/10.1145/3548606.3559367.

[174] X. Zhao, H. Qu, J. Xu, S. Li och G.-G. Wang. “AMSFuzz: An adaptive mutation schedule for fuzzing”. I: *Expert Systems with Applications* 208 (dec. 2022), s. 118162. https://doi.org/10.1016/j.eswa.2022.118162.

[175] T. R. Devine, M. Campbell, M. Anderson och D. Dzielski. “SREP+SAST: A Comparison of Tools for Reverse Engineering Machine Code to Detect Cybersecurity Vulnerabilities in Binary Executables”. I: *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, dec. 2022, s. 862–869. https://doi.org/10.1109/csci58124.2022.00156.

[176] M. Ma, L. Han och Y. Qian. “CVDF DYNAMIC—A Dynamic Fuzzy Testing Sample Generation Framework Based on BI-LSTM and Genetic Algorithm”. I: *Sensors* 22.3 (febr. 2022), s. 1265. https://doi.org/10.3390/s22031265.

[177] L. Reichling, I. Warsame, S. Reilly, A. Brownfield, N. Niu och B. Wang. “FaultHunter: Automatically Detecting Vulnerabilities in C against Fault Injection Attacks”. I: *2022 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*. IEEE, dec. 2022, s. 271–276. https://doi.org/10.1109/bdcat56447.2022.00045.

[178] L. He, H. Hu, P. Su, Y. Cai och Z. Liang. “FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries”. I: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, aug. 2022, s. 2497–2512. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/he-liang.

[179] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang och C. Wu. “Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths”. I: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, aug. 2022, s. 4365–4382. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhou-shunfan.

[180] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao och H. Jin. “mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations”. I: *IEEE Transactions on Dependable and Secure Computing* (2024), s. 1–12. https://doi.org/10.1109/tdsc.2022.3199769.

[181] J. Chen, J. Wang, C. Song och H. Yin. “JIGSAW: Efficient and Scalable Path Constraints Fuzzing”. I: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2022, s. 18–35. https://doi.org/10.1109/sp46214.2022.9833796.

[182] K.-T. Xie, J.-J. Bai, Y.-H. Zou och Y.-P. Wang. “ROZZ: Property-based Fuzzing for Robotic Programs in ROS”. I: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, maj 2022, s. 6786–6792. https://doi.org/10.1109/icra46639.2022.9811701.

[183] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu och J. Sun. “PATA: Fuzzing with Path Aware Taint Analysis”. I: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2022, s. 1–17. https://doi.org/10.1109/sp46214.2022.9833594.

[184] X. Zhu, S. Wen, A. Jolfaei, M. S. Haghighi, S. Camtepe och Y. Xiang. “Vulnerability Detection in SIoT Applications: A Fuzzing Method on their Binaries”. I: *IEEE Transactions on Network Science and Engineering* 9.3 (maj 2022), s. 970–979. https://doi.org/10.1109/tnse.2020.3038142.

[185] N. Ruaro, K. Zeng, L. Dresel, M. Polino, T. Bao, A. Continella, S. Zanero, C. Kruegel och G. Vigna. “SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning”. I: *24th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’21. ACM, okt. 2021, s. 456–468. https://doi.org/10.1145/3471621.3471865.

[186] M. Ren, X. Ren, H. Feng, J. Ming och Y. Lei. “Z-Fuzzer: device-agnostic fuzzing of Zigbee protocol implementation”. I: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec ’21. ACM, juni 2021, s. 347–358. https://doi.org/10.1145/3448300.3468296.

[187] A. Aumpansub och Z. Huang. “Detecting Software Vulnerabilities Using Neural Networks”. I: *2021 13th International Conference on Machine Learning and Computing*. ICMLC 2021. ACM, febr. 2021, s. 166–171. https://doi.org/10.1145/3457682.3457707.

[188] X. Liu, T. Sun, Z. Zhao, D. Wu, Q. Hu, W. Chen och L. Gu. “IfCut: If Branch Cutting Based Fuzzing System”. I: *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE, okt. 2021, s. 389–395. https://doi.org/10.1109/dsc53577.2021.00061.

[189] Y. Zhang, Z. Wang, W. Yu och B. Fang. “Multi-level Directed Fuzzing for Detecting Use-after-Free Vulnerabilities”. I: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, okt. 2021, s. 569–576. https://doi.org/10.1109/trustcom53373.2021.00087.

[190] C. T. Tran och S. Kurmangaleev. “Futag: Automated fuzz target generator for testing software libraries”. I: *2021 Ivannikov Memorial Workshop (IVMEM)*. IEEE, sept. 2021. https://doi.org/10.1109/ivmem53963.2021.00021.

[191] Y. Iqbal, M. A. Sindhu, M. H. Arif och M. A. Javed. “Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool”. I: *2021 International Conference on Cyber Warfare and Security (ICCWS)*. IEEE, nov. 2021, s. 112–117. https://doi.org/10.1109/iccws53234.2021.9703043.

[192] B. Swain, B. Liu, P. Liu, Y. Li, A. Crump, R. Khera och J. Huang. “OpenRace: An Open Source Framework for Statically Detecting Data Races”. I: *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, nov. 2021, s. 25–32. https://doi.org/10.1109/correctness54621.2021.00009.

[193] Z. Li, H. Washizaki och Y. Fukazawa. “Feature Extraction Method for Cross-Architecture Binary Vulnerability Detection”. I: *2021 IEEE 10th Global Conference on Consumer Electronics (GCCE)*. IEEE, okt. 2021, s. 834–836. https://doi.org/10.1109/gcce53005.2021.9621783.

[194] K. M. Alshmrany, M. Aldughaim, A. Bhayat och L. C. Cordeiro. “FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs”. I: *Tests and Proofs*. Springer International Publishing, 2021, s. 85–105. https://doi.org/10.1007/978-3-030-79379-1_6.

[195] X. Wang, C. Hu, R. Ma, D. Tian och J. He. “CMFuzz: context-aware adaptive mutation for fuzzers”. I: *Empirical Software Engineering* 26.1 (jan. 2021). https://doi.org/10.1007/s10664-020-09927-3.

[196] Á. J. Varela-Vaca, R. M. Gasca, J. A. Carmona-Fombella och M. T. Gómez-López. “AMADEUS: towards the AutoMAteD secUrity teSting”. I: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC ’20. ACM, okt. 2020, s. 1–12. https://doi.org/10.1145/3382025.3414952.

[197] S. Yan, C. Wu, H. Li, W. Shao och C. Jia. “PathAFL: Path-Coverage Assisted Fuzzing”. I: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’20. ACM, okt. 2020, s. 598–609. https://doi.org/10.1145/3320269.3384736.

[198] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao och J. Sun. “EM-Fuzz: Augmented Firmware Fuzzing via Memory Checking”. I: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (nov. 2020), s. 3420–3432. https://doi.org/10.1109/tcad.2020.3013046.

[199] J. Chen, J. Chen, D. Guo och D. Towey. “An improved fuzzing approach based on adaptive random testing”. I: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, okt. 2020, s. 103–108. https://doi.org/10.1109/issrew51248.2020.00045.

[200] X. Wang, C. Hu, R. Ma, B. Li och X. Wang. “LAFuzz: Neural Network for Efficient Fuzzing”. I: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, nov. 2020, s. 603–611. https://doi.org/10.1109/ictai50040.2020.00098.

[201] G. Lin, S. Wen, Q.-L. Han, J. Zhang och Y. Xiang. “Software Vulnerability Detection Using Deep Neural Networks: A Survey”. I: *Proceedings of the IEEE* 108.10 (okt. 2020), s. 1825–1848. https://doi.org/10.1109/jproc.2020.2993293.

[202] R. K. Medicherla, R. Komondoor och A. Roychoudhury. “Fitness Guided Vulnerability Detection with Greybox Fuzzing”. I: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ICSE ’20. ACM, juni 2020, s. 513–520. https://doi.org/10.1145/3387940.3391457.

[203] X. Mi, B. Wang, Y. Tang, P. Wang och B. Yu. “SHFuzz: Selective Hybrid Fuzzing with Branch Scheduling Based on Binary Instrumentation”. I: *Applied Sciences* 10.16 (aug. 2020), s. 5449. https://doi.org/10.3390/app10165449.

[204] F. Rustamov, J. Kim och J. Yun. “DeepDiver: Diving into Abysmal Depth of the Binary for Hunting Deeply Hidden Software Vulnerabilities”. I: *Future Internet* 12.4 (april 2020), s. 74. https://doi.org/10.3390/fi12040074.

[205] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei och L. Lu. “SAVIOR: Towards Bug-Driven Hybrid Testing”. I: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2020, s. 1580–1596. https://doi.org/10.1109/sp40000.2020.00002.

[206] I: *Proceedings of 2020 the 10th International Workshop on Computer Science and Engineering*. WCSE 2020. WCSE, 2020. https://doi.org/10.18178/wcse.2020.06.008.

[207] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang och Y. Liu. “MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs”. I: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC’20. USA: USENIX Association, 2020.

[208] S. Singh och S. Khurshid. “Parallel Chopped Symbolic Execution”. I: *Formal Methods and Software Engineering*. Springer International Publishing, 2020, s. 107–125. https://doi.org/10.1007/978-3-030-63406-3_7.

[209] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz och M. Lemerre. “Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities”. I: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, okt. 2020, s. 47–62. URL: `https://www.usenix.org/conference/raid2020/presentation/nguyen`.

[210] K. K. Ispoglou, D. Austin, V. Mohan och M. Payer. “FuzzGen: automatic fuzzer generation”. I: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC’20. USA: USENIX Association, 2020.

[211] N. Vinesh och M. Sethumadhavan. “ConFuzz—A Concurrency Fuzzer”. I: *First International Conference on Sustainable Technologies for Computational Intelligence*. Springer Singapore, nov. 2019, s. 667–691. https://doi.org/10.1007/978-981-15-0029-9_53.

[212] Y. Wang, Z. Wu, Q. Wei och Q. Wang. “Field-aware Evolutionary Fuzzing Based on Input Specifications and Vulnerability Metrics”. I: *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, okt. 2019. https://doi.org/10.1109/icsess47205.2019.9040809.

[213] J. Jiang, X. Yu, Y. Sun och H. Zeng. “A Survey of the Software Vulnerability Discovery Using Machine Learning Techniques”. I: *Artificial Intelligence and Security*. Springer International Publishing, 2019, s. 308–317. https://doi.org/10.1007/978-3-030-24268-8_29.

[214] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu och L. Sun. “FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation”. I: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, aug. 2019, s. 1099–1114. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/zheng`.

[215] A. Al-Nusirat, F. Hanandeh, M. K. Kharabsheh, M. Al-Ayyoub och N. Al-dhfairi. “Dynamic Detection of Software Defects Using Supervised Learning Techniques”. I: *International Journal of Communication Networks and Information Security (IJCNIS)* 11.1 (april 2022). https://doi.org/10.17762/ijcnis.v11i1.3966.

[216] Z. Xu och G. Liu. “STACKEEPER: A Static Source Code Analyzer to Detect Stack-based Uninitialized Use Vulnerabilities”. I: *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. IEEE, dec. 2018, s. 2180–2184. https://doi.org/10.1109/compcomm.2018.8780675.

[217] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu och Y. Liu. “Hawkeye: Towards a Desired Directed Grey-box Fuzzer”. I: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. ACM, okt. 2018, s. 2095–2108. https://doi.org/10.1145/3243734.3243849.

[218] J. Jurn, T. Kim och H. Kim. “An Automated Vulnerability Detection and Remediation Method for Software Security”. I: *Sustainability* 10.5 (maj 2018), s. 1652. https://doi.org/10.3390/su10051652.

[219] D. Beyer och T. Lemberger. “Software Verification: Testing vs. Model Checking”. I: *Hardware and Software: Verification and Testing*. Springer International Publishing, 2017, s. 99–114. https://doi.org/10.1007/978-3-319-70389-3_7.

[220] B. Chen, Q. Zeng och W. Wang. “Crashmaker: an improved binary concolic testing tool for vulnerability detection”. I: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC 2014. ACM, mars 2014, s. 1257–1263. https://doi.org/10.1145/2554850.2554875.

[221] E. H. Boudjema, S. Verlan, L. Mokdad och C. Faure. “VYPER: Vulnerability detection in binary code”. I: *SECURITY AND PRIVACY* 3.2 (dec. 2019). https://doi.org/10.1002/spy2.100.

[222] S. Jeon och H. K. Kim. “AutoVAS: An automated vulnerability analysis system with a deep learning approach”. I: *Computers & Security* 106 (juli 2021), s. 102308. https://doi.org/10.1016/j.cose.2021.102308.

[223] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel och G. Vigna. “Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices”. I: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2021, s. 484–500. https://doi.org/10.1109/sp40001.2021.00066.

[224] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang och Y. Wu. “VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities”. I: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. IJCAI-2019. International Joint Conferences on Artificial Intelligence Organization, aug. 2019, s. 4665–4671. https://doi.org/10.24963/ijcai.2019/648.

[225] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu och Z. Chen. “SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities”. I: *IEEE Transactions on Dependable and Secure Computing* 19.4 (juli 2022), s. 2244–2258. https://doi.org/10.1109/tdsc.2021.3051525.

[226] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng och Y. Zhong. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection”. I: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. https://doi.org/10.14722/ndss.2018.23158.

[227] Z. Chen, C. Wang, J. Yan, Y. Sui och J. Xue. “Runtime detection of memory errors with smart status”. I: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’21. ACM, juli 2021, s. 296–308. https://doi.org/10.1145/3460319.3464807.

[228] X. Zhao, H. Qu, W. Lv, S. Li och J. Xu. “MooFuzz: Many-Objective Optimization Seed Schedule for Fuzzer”. I: *Mathematics* 9.3 (jan. 2021), s. 205. https://doi.org/10.3390/math9030205.

[229] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian och B. Liang. “SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits”. I: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, okt. 2017, s. 2139–2154. https://doi.org/10.1145/3133956.3134085.

[230] J. Ye, B. Zhang, R. Li, C. Feng och C. Tang. “Program State Sensitive Parallel Fuzzing for Real World Software”. I: *IEEE Access* 7 (2019), s. 42557–42564. https://doi.org/10.1109/access.2019.2905744.

[231] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang och B. Liang. “ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery”. I: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2019, s. 769–786. https://doi.org/10.1109/sp.2019.00057.

[232] Z. Song, J. Wang, S. Liu, Z. Fang och K. Yang. “HGVul: A Code Vulnerability Detection Method Based on Heterogeneous Source-Level Intermediate Representation”. I: *Security and Communication Networks* 2022 (april 2022). Utg. av G. Zhaoquan, s. 1–13. https://doi.org/10.1155/2022/1919907.

[233] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe och M. Payer. “FirmFuzz: Automated IoT Firmware Introspection and Analysis”. I: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. CCS ’19. ACM, nov. 2019. https://doi.org/10.1145/3338507.3358616.

[234] W. Tang, M. Tang, M. Ban, Z. Zhao och M. Feng. “CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection”. I: *Journal of Systems and Software* 199 (maj 2023), s. 111623. https://doi.org/10.1016/j.jss.2023.111623.

[235] H. Huang, P. Yao, R. Wu, Q. Shi och C. Zhang. “Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction”. I: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2020, s. 1613–1627. https://doi.org/10.1109/sp40000.2020.00063.

[236] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal och Y. Xiang. “Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference”. I: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. ACM, nov. 2021. https://doi.org/10.1145/3460120.3484543.

[237] S. B. Mazzone, M. Pagnozzi, A. Fattori, A. Reina, A. Lanzi och D. Bruschi. “Improving Mac OS X security through gray box fuzzing technique”. I: *Proceedings of the Seventh European Workshop on System Security*. EuroSys 2014. ACM, april 2014, s. 1–6. https://doi.org/10.1145/2592791.2592793.

[238] W. Luo och B. Demsky. “C11Tester: a race detector for C/C++ atomics”. I: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’21. ACM, april 2021. https://doi.org/10.1145/3445814.3446711.

[239] A. Ye, L. Wang, L. Zhao, J. Ke, W. Wang och Q. Liu. “RapidFuzz: Accelerating fuzzing via Generative Adversarial Networks”. I: *Neurocomputing* 460 (okt. 2021), s. 195–204. https://doi.org/10.1016/j.neucom.2021.06.082.

[240] M. Xu, S. Kashyap, H. Zhao och T. Kim. “Krace: Data Race Fuzzing for Kernel File Systems”. I: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2020. https://doi.org/10.1109/sp40000.2020.00078.

[241] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang och A. Hanna. “BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices”. I: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2018, s. 114–138. https://doi.org/10.1007/978-3-319-93411-2_6.

[242] I. Gotovchits, R. Van Tonder och D. Brumley. “Saluki: Finding Taint-style Vulnerabilities with Static Property Checking”. I: *Proceedings 2018 Workshop on Binary Analysis Research*. BAR 2018. Internet Society, 2018. https://doi.org/10.14722/bar.2018.23019.

[243] Y. Ko, B. Zhu och J. Kim. “Fuzzing with automatically controlled interleavings to detect concurrency bugs”. I: *Journal of Systems and Software* 191 (sept. 2022), s. 111379. https://doi.org/10.1016/j.jss.2022.111379.

[244] V.-T. Pham, M. Bohme och A. Roychoudhury. “AFLNET: A Greybox Fuzzer for Network Protocols”. I: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, okt. 2020, s. 460–465. https://doi.org/10.1109/icst46399.2020.00062.

[245] M. Xu, C. Qian, K. Lu, M. Backes och T. Kim. “Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels”. I: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2018. https://doi.org/10.1109/sp.2018.00017.

[246] C. Lidbury och A. F. Donaldson. “Dynamic race detection for C++11”. I: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. ACM, jan. 2017, s. 443–457. https://doi.org/10.1145/3009837.3009857.

[247] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee och I. Shin. “Razzer: Finding Kernel Race Bugs through Fuzzing”. I: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2019, s. 754–768. https://doi.org/10.1109/sp.2019.00017.

[248] M. Cui, C. Chen, H. Xu och Y. Zhou. “SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis”. I: *ACM Transactions on Software Engineering and Methodology* 32.4 (maj 2023), s. 1–21. https://doi.org/10.1145/3542948.

[249] G. Grieco, M. Ceresa och P. Buiras. “QuickFuzz: an automatic random fuzzer for common file formats”. I: *Proceedings of the 9th International Symposium on Haskell*. ICFP’16. ACM, sept. 2016, s. 13–20. https://doi.org/10.1145/2976002.2976017.

[250] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran och D. Phung. “ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection”. I: *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, maj 2022, s. 178–182. https://doi.org/10.1109/icse-companion55297.2022.9793807.

[251] N. Emamdoost, Q. Wu, K. Lu och S. McCamant. “Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning”. I: *Proceedings 2021 Network and Distributed System Security Symposium*. NDSS 2021. Internet Society, 2021. https://doi.org/10.14722/ndss.2021.24416.

[252] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian och Z. Wang. “Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection”. I: *IEEE Transactions on Information Forensics and Security* 16 (2021), s. 1943–1958. https://doi.org/10.1109/tifs.2020.3044773.

[253] H. Li, H. Kwon, J. Kwon och H. Lee. “A Scalable Approach for Vulnerability Discovery Based on Security Patches”. I: *Applications and Techniques in Information Security*. Springer Berlin Heidelberg, 2014, s. 109–122. https://doi.org/10.1007/978-3-662-45670-5_11.

[254] Y. Cai, P. Yao och C. Zhang. “Canary: practical static detection of inter-thread value-flow bugs”. I: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI ’21. ACM, juni 2021. https://doi.org/10.1145/3453483.3454099.

[255] V.-T. Pham, M. Böhme och A. Roychoudhury. “Model-based whitebox fuzzing for program binaries”. I: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE’16. ACM, aug. 2016, s. 543–553. https://doi.org/10.1145/2970276.2970316.

[256] T. Li, J.-J. Bai, Y. Sui och S.-M. Hu. “Path-sensitive and alias-aware typestate analysis for detecting OS bugs”. I: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’22. ACM, febr. 2022, s. 859–872. https://doi.org/10.1145/3503222.3507770.

[257] D. Gens, S. Schmitt, L. Davi och A.-R. Sadeghi. “K-Miner: Uncovering Memory Corruption in Linux”. I: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. https://doi.org/10.14722/ndss.2018.23326.

[258] G. Jie, K. Xiao-Hui och L. Qiang. “Survey on Software Vulnerability Analysis Method Based on Machine Learning”. I: *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*. IEEE, juni 2016, s. 642–647. https://doi.org/10.1109/dsc.2016.33.

[259] X. Li, L. Sun, R. Jiang, H. Qu och Z. Yan. “OTA: An Operation-oriented Time Allocation Strategy for Greybox Fuzzing”. I: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mars 2021, s. 108–118. https://doi.org/10.1109/saner50967.2021.00019.

[260] J. Akram, Z. Shi, M. Mumtaz och P. Luo. “DCCD: An Efficient and Scalable Distributed Code Clone Detection Technique for Big Code”. I: *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*. Vol. 2018. SEKE2018. KSI Research Inc. och Knowledge Systems Institute Graduate School, juli 2018, s. 354–390. https://doi.org/10.18293/seke2018-117.

[261] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin och B. Lee. “HFL: Hybrid Fuzzing on the Linux Kernel”. I: *Proceedings 2020 Network and Distributed System Security Symposium*. NDSS 2020. Internet Society, 2020. https://doi.org/10.14722/ndss.2020.24018.

[262] M. Böhme, V.-T. Pham, M.-D. Nguyen och A. Roychoudhury. “Directed Greybox Fuzzing”. I: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, okt. 2017, s. 2329–2344. https://doi.org/10.1145/3133956.3134020.

[263] H. Hanif och S. Maffeis. “VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection”. I: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, juli 2022, s. 1–8. https://doi.org/10.1109/ijcnn55064.2022.9892280.

[264] Z. Gui, H. Shu, F. Kang och X. Xiong. “FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution”. I: *IEEE Access* 8 (2020), s. 29826–29841. https://doi.org/10.1109/access.2020.2973043.

[265] J. Gao, X. Yang, Y. Fu, Y. Jiang och J. Sun. “VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary”. I: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE ’18. ACM, sept. 2018. https://doi.org/10.1145/3238147.3240480.

[266] X. Li, L. Wang, Y. Xin, Y. Yang, Q. Tang och Y. Chen. “Automated Software Vulnerability Detection Based on Hybrid Neural Network”. I: *Applied Sciences* 11.7 (april 2021), s. 3201. https://doi.org/10.3390/app11073201.

[267] S. Cao, X. Sun, L. Bo, Y. Wei och B. Li. “BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection”. I: *Information and Software Technology* 136 (aug. 2021), s. 106576. https://doi.org/10.1016/j.infsof.2021.106576.

[268] Y. Yao, W. Zhou, Y. Jia, L. Zhu, P. Liu och Y. Zhang. “Identifying Privilege Separation Vulnerabilities in IoT Firmware with Symbolic Execution”. I: *Computer Security – ESORICS 2019*. Springer International Publishing, 2019, s. 638–657. https://doi.org/10.1007/978-3-030-29959-0_31.

[269] Y. Hu och I. Neamtiu. “Static Detection of Event-based Races in Android Apps”. I: *ACM SIGPLAN Notices* 53.2 (mars 2018), s. 257–270. https://doi.org/10.1145/3296957.3173173.

[270] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida och H. Bos. “CollabFuzz: A Framework for Collaborative Fuzzing”. I: *Proceedings of the 14th European Workshop on Systems Security*. EuroSys ’21. ACM, april 2021, s. 1–7. https://doi.org/10.1145/3447852.3458720.

[271] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang och P. Su. “HTFuzz: Heap Operation Sequence Sensitive Fuzzing”. I: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. ACM, okt. 2022. https://doi.org/10.1145/3551349.3560415.

[272] J. Wang, B. Chen, L. Wei och Y. Liu. “Skyfire: Data-Driven Seed Generation for Fuzzing”. I: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2017, s. 579–594. https://doi.org/10.1109/sp.2017.23.

[273] H. Yan, S. Luo, L. Pan och Y. Zhang. “HAN-BSVD: A hierarchical attention network for binary software vulnerability detection”. I: *Computers & Security* 108 (sept. 2021), s. 102286. https://doi.org/10.1016/j.cose.2021.102286.

[274] S. Schumilo, C. Aschermann, A. Abbasi, S. Worner och T. Holz. “HYPER-CUBE: High-Dimensional Hypervisor Fuzzing”. I: *Proceedings 2020 Network and Distributed System Security Symposium*. NDSS 2020. Internet Society, 2020. https://doi.org/10.14722/ndss.2020.23096.

[275] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida och T. Holz. “Cupid : Automatic Fuzzer Selection for Collaborative Fuzzing”. I: *Annual Computer Security Applications Conference*. ACSAC ’20. ACM, dec. 2020, s. 360–372. https://doi.org/10.1145/3427228.3427266.

[276] H. Li, H. Kwon, J. Kwon och H. Lee. “CLORIFI: software vulnerability discovery using code clone verification”. I: *Concurrency and Computation: Practice and Experience* 28.6 (juni 2015), s. 1900–1917. https://doi.org/10.1002/cpe.3532.

[277] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei och Z. Chen. “CollAFL: Path Sensitive Fuzzing”. I: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, maj 2018, s. 679–696. https://doi.org/10.1109/sp.2018.00040.

[278] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang och Z. Wang. “Automated conformance testing for JavaScript engines via deep compiler fuzzing”. I: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI ’21. ACM, juni 2021, s. 435–450. https://doi.org/10.1145/3453483.3454054.

[279] Z.-M. Jiang, J.-J. Bai, K. Lu och S.-M. Hu. “Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection”. I: *Proceedings 2022 Network and Distributed System Security Symposium*. NDSS 2022. Internet Society, 2022. https://doi.org/10.14722/ndss.2022.24296.

Bilaga A – Granskningsprotokoll

Tabell A.1. Granskningsprotokoll för fulltextgranskning.

Source, Authors, Title, Year, Source Title, Volume, Issue, Art.no., Page start, Page end, Link, Abstract, Document type	Från databas (Web of Science, Scopus)
Relevans 1	Relevansskattning, granskare 1
Relevans 2	Relevansskattning, granskare 2
Relevans sammanlagd	Summerad relevansskattning
Fulltextgranskare	Granskarens namn
Fulltext motiv exkludera	Exkluderingskriterierna + ev. fritext
Bedöms publikationen användbar för bakgrund, till exempel en sekundär- eller tertiärstudie?	Ja, Nej
Beskriver publikationen en taxonomi för mjukvarusårbarheter?	Ja, Nej
Beskriver publikationen konkreta tekniker för att identifiera mjukvarusårbarheter?	Ja, Nej
Beskriver publikationen konkreta verktyg för att identifiera mjukvarusårbarheter?	Ja, Nej
Vilken kategori av teknik eller verktyg beskriver publikationen?	Taxonomikategorier + fritext för hybrid
För vilka programspråk är tekniken eller verktyget relevant?	fritext
Hur mogen bedöms tekniken eller verktygen som beskrivs i publikationen?	TRL-skala 1-9
Beskriver publikationen hur tekniken eller verktyget har testats?	Ja, Nej
Finns verktyget eller kod till ett verktyg tillgängligt, till exempel via ett gitrepo eller motsvarande?	Ja, Nej + fritext (länk)
Finns testcase som använts tillgängliga, till exempel via ett gitrepo eller motsvarande?	Ja, Nej + fritext(länk)
Bedöms arbetet reproducerbart?	Ja, Nej
Författarens slutsatser i publikationen	Fritext, granskarens summering
Granskarens slutsatser av publikationen	Fritext, granskarens summering
Generella kommentarer	Fritext

Bilaga B – Resultat av fulltextgranskning

I tabell B.1 finns en förteckning över de publikationer som har TRL-skattats under fulltextgranskningen.

Tabell B.1. Granskade verktyg i fulltextanalys.

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[51]	2021	Mönsterigenkänning	6	C, C++
[52]	2015	Hybridmetoder	6	-
[53]	2016	Statisk analys	3	LLVM
[54]	2024	Mönsterigenkänning	7	Python
[55]	2020	Mönsterigenkänning	4	Java
[56]	2019	Statisk analys	5	Java
[57]	2023	Mönsterigenkänning	6	-
[58]	2021	Mönsterigenkänning	5	Binärer
[59]	2019	Hybridmetoder	5	Java
[60]	2017	Statisk analys	3	Binärer
[61]	2016	Statisk analys	3	Java
[62]	2020	Mönsterigenkänning	4	C, C++
[63]	2015	Statisk analys	5	Binärer
[64]	2017	Statisk analys	4	C
[65]	2018	Hybridmetoder	5	C, C++
[66]	2018	Statisk analys	6	C, C++
[67]	2021	Statisk analys	7	Binärer
[68]	2024	Statisk analys	6	C, C++
[69]	2019	Dynamisk analys	7	C, C++
[70]	2020	Hybridmetoder	6	-
[71]	2019	Statisk analys	6	C
[72]	2021	Statisk analys	7	Binärer
[73]	2017	Fuzzer	7	Binärer
[74]	2022	Fuzzer	7	-
[75]	2023	Mönsterigenkänning	7	LLVM
[76]	2016	Statisk analys	5	Android
[77]	2023	Mönsterigenkänning	5	C, C++
[78]	2022	Fuzzer	7	CLI-baserade program
[79]	2022	Fuzzer	7	IoT
[80]	2022	Dynamisk analys	6	-
[81]	2020	Hybridmetoder	6	-
[82]	2014	Dynamisk analys	5	-
[83]	2016	Hybridmetoder	6	-
[84]	2019	Fuzzer	7	Binärer
[85]	2016	Dynamisk analys	6	C, C++
[86]	2023	Statisk analys	6	Binärer
[87]	2022	Hybridmetoder	7	C, C++
[88]	2021	Statisk analys	7	Rust
[89]	2018	Hybridmetoder	6	Binärer
[90]	2022	Hybridmetoder	6	C, C++
[91]	2019	Modellbaserad testning	6	Binärer
[92]	2022	Mönsterigenkänning	3	LLVM

Tabell B.1. Granskade verktyg i fulltextanalys (forts.).

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[93]	2019	Hybridmetoder	6	-
[94]	2023	Hybridmetoder	6	C
[95]	2016	Dynamisk analys	3	Java
[96]	2023	Mönsterigenkänning	6	C, C++
[97]	2022	Hybridmetoder	5	-
[98]	2020	Statisk analys	4	C, C++
[99]	2021	Hybridmetoder	6	Java
[100]	2018	Statisk analys	5	Java
[101]	2020	Statisk analys	6	C, C++
[102]	2021	Statisk analys	7	C, C++
[103]	2022	Statisk analys	7	Rust, C, C++
[104]	2015	Dynamisk analys	5	Inbyggd mjukvara
[105]	2024	Mönsterigenkänning	7	C, C++
[106]	2015	Statisk analys	4	C, C++
[107]	2023	Mönsterigenkänning	6	C, C++
[108]	2022	Statisk analys	6	Android
[36]	2016	Hybridmetoder	8	LLVM, binärer
[109]	2022	Mönsterigenkänning	3	C, C++
[110]	2022	Fuzzer	5	-
[111]	2017	Mönsterigenkänning	4	C, C++
[112]	2015	Statisk analys	5	C
[113]	2018	Dynamisk analys	6	C
[114]	2024	Statisk analys	7	Java
[115]	2019	Fuzzer	4	-
[116]	2018	Hybridmetoder	7	Java
[117]	2016	Statisk analys	6	C, C++
[118]	2022	Mönsterigenkänning	7	C, C++
[119]	2022	Statisk analys	7	LLVM
[120]	2022	Statisk analys	4	C, C++
[121]	2021	Statisk analys	5	C, C++
[28]	2021	Mönsterigenkänning	5	Java
[122]	2021	Statisk analys	6	Inbyggd mjukvara
[123]	2024	Fuzzer	6	-
[124]	2020	Statisk analys	7	Binärer, C++
[125]	2023	Dynamisk analys	7	C
[126]	2024	Fuzzer	6	-
[127]	2020	Statisk analys	5	-
[128]	2018	Statisk analys	3	C
[129]	2023	Statisk analys	6	Inbyggd mjukvara
[130]	2015	Fuzzer	5	-
[131]	2017	Statisk analys	4	Java, Android
[132]	2021	Fuzzer	6	-
[133]	2018	Fuzzer	3	-
[134]	2023	Hybridmetoder	4	-
[135]	2018	Statisk analys	5	Inbyggd mjukvara
[136]	2023	Statisk analys	3	C, C++
[137]	2023	Hybridmetoder	7	Java

Tabell B.1. Granskade verktyg i fulltextanalys (forts.).

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[138]	2020	Fuzzer	6	-
[139]	2022	Hybridmetoder	5	-
[140]	2023	Statisk analys	6	-
[141]	2024	Mönsterigenkänning	4	Java
[142]	2020	Fuzzer	6	-
[143]	2019	Fuzzer	5	Java
[144]	2019	Fuzzer	7	-
[145]	2017	Hybridmetoder	3	Binärer
[29]	2021	Fuzzer	8	-
[146]	2019	Fuzzer	6	-
[147]	2023	Fuzzer	4	C, C++
[148]	2017	Fuzzer	5	-
[149]	2024	Fuzzer	6	-
[150]	2019	Hybridmetoder	5	-
[151]	2018	Fuzzer	3	-
[152]	2017	Modellbaserad testning	4	Windows
[153]	2022	Fuzzer	3	-
[154]	2024	Fuzzer	3	Konfigurationsparametrar
[155]	2024	Mönsterigenkänning	2	-
[156]	2023	Fuzzer	4	Python
[157]	2023	Statisk analys	5	C, C++
[158]	2023	Hybridmetoder	6	C, C++
[159]	2023	Mönsterigenkänning	4	C, C++
[160]	2023	Statisk analys	3	-
[161]	2023	Fuzzer	7	MacOS
[162]	2023	Formella metoder	5	C
[163]	2023	Fuzzer	5	-
[164]	2023	Fuzzer	6	-
[165]	2023	Fuzzer	3	-
[166]	2023	Fuzzer	6	-
[167]	2023	Statisk analys	7	Binärer
[35]	2023	Fuzzer	9	-
[168]	2023	Mönsterigenkänning	6	Binärer
[169]	2023	Fuzzer	6	C
[170]	2023	Mönsterigenkänning	4	Python
[171]	2023	Mönsterigenkänning	4	Javascript
[172]	2023	Hybridmetoder	7	Binärer, IoT
[173]	2022	Fuzzer	7	IoT
[174]	2022	Fuzzer	5	-
[175]	2022	Statisk analys	3	Binärer
[176]	2022	Hybridmetoder	5	-
[177]	2022	Statisk analys	5	C
[178]	2022	Statisk analys	5	C, C++
[179]	2022	Dynamisk analys	7	-
[180]	2022	Modellbaserad testning	7	-
[41]	2022	Hybridmetoder	8	C, C++
[181]	2022	Fuzzer	7	-
[34]	2022	Fuzzer	8	Inbyggd mjukvara
[182]	2022	Fuzzer	7	Robot Operating System (ROS)

Tabell B.1. Granskade verktyg i fulltextanalys (forts.).

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[183]	2022	Fuzzer	7	-
[184]	2022	Fuzzer	7	Binärer
[185]	2021	Hybridmetoder	5	-
[186]	2021	Fuzzer	7	IoT
[187]	2021	Statisk analys	5	C, C++
[188]	2021	Fuzzer	5	C
[189]	2021	Fuzzer	7	C, C++
[39]	2021	Fuzzer	8	C, C++
[190]	2021	Fuzzer	7	Programbibliotek
[191]	2021	Dynamisk analys	6	C, C++
[192]	2021	Statisk analys	7	LLVM, Rust, haskell, kotlin
[193]	2021	Mönsterigenkänning	4	Binärer
[194]	2021	Hybridmetoder	7	C
[195]	2021	Fuzzer	5	-
[196]	2020	Modellbaserad testning	5	-
[197]	2020	Fuzzer	7	-
[198]	2020	Fuzzer	5	Inbyggd mjukvara
[199]	2020	Fuzzer	4	C, C++
[200]	2020	Fuzzer	5	-
[201]	2020	Hybridmetoder	4	-
[202]	2020	Fuzzer	4	-
[203]	2020	Hybridmetoder	5	-
[204]	2020	Hybridmetoder	4	-
[205]	2020	Hybridmetoder	5	-
[206]	2020	Fuzzer	4	-
[207]	2020	Fuzzer	6	Multitrådade program
[208]	2020	Dynamisk analys	4	-
[209]	2020	Fuzzer	5	-
[210]	2020	Fuzzer	5	-
[211]	2020	Fuzzer	4	-
[212]	2019	Fuzzer	3	-
[213]	2019	Hybridmetoder	5	Maskininlärning
[214]	2019	Fuzzer	5	IoT
[215]	2019	Mönsterigenkänning	3	Maskininlärning
[216]	2018	Statisk analys	3	C, C++
[217]	2018	Fuzzer	4	-
[218]	2018	Hybridmetoder	3	-
[37]	2018	Dynamisk analys	8	Inbyggd mjukvara, C
[219]	2017	Hybridmetoder	1	-
[220]	2014	Dynamisk analys	4	-
[221]	2019	Dynamisk analys	7	Binärer
[222]	2021	Hybridmetoder	7	C, C++
[223]	2021	Hybridmetoder	7	IoT
[224]	2019	Statisk analys	5	C, C++
[225]	2021	Statisk analys	6	C, C++
[226]	2018	Statisk analys	5	C, C++
[227]	2021	Dynamisk analys	6	C

Tabell B.1. Granskade verktyg i fulltextanalys (forts.).

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[228]	2021	Fuzzer	6	C, C++
[229]	2017	Fuzzer	4	Neurolingvistisk programmering
[230]	2019	Fuzzer	5	C
[231]	2019	Fuzzer	5	Binärer
[232]	2022	Hybridmetoder	5	C, C++
[233]	2019	Hybridmetoder	5	IoT, Inbyggd mjukvara
[234]	2023	Mönsterigenkänning	5	Binärer
[235]	2020	Hybridmetoder	5	-
[236]	2021	Fuzzer	6	IoT, Inbyggd mjukvara
[237]	2014	Fuzzer	4	MacOS
[238]	2021	Statisk analys	5	C, C++
[239]	2021	-	5	C
[240]	2020	Fuzzer	5	-
[241]	2018	-	5	Inbyggd mjukvara
[242]	2018	Statisk analys	5	Binärer
[243]	2022	Hybridmetoder	5	C
[244]	2020	Fuzzer	7	-
[245]	2018	Statisk analys	5	C
[246]	2016	Dynamisk analys	6	C, C++
[247]	2019	Hybridmetoder	6	C
[248]	2022	Statisk analys	6	Rust
[249]	2016	Fuzzer	6	-
[250]	2022	Mönsterigenkänning	7	-
[251]	2021	Statisk analys	5	C
[252]	2020	Mönsterigenkänning	6	C, Java, Swift, Php
[253]	2014	Hybridmetoder	4	-
[254]	2021	Statisk analys	5	C, C++
[255]	2016	Fuzzer	4	Binärer
[256]	2022	Statisk analys	5	C
[257]	2018	Statisk analys	6	C, C++
[258]	2017	Mönsterigenkänning	4	-
[259]	2021	Fuzzer	5	C
[260]	2018	Statisk analys	5	Java, JavaScript, C, C++, C#, Xml, Python
[261]	2020	Hybridmetoder	5	C
[262]	2017	Fuzzer	7	-
[263]	2022	Mönsterigenkänning	4	C, C++
[264]	2020	Fuzzer	7	Inbyggd mjukvara, IoT
[265]	2018	Mönsterigenkänning	5	Binärer
[266]	2021	Mönsterigenkänning	5	C
[267]	2021	Mönsterigenkänning	5	C, C++
[268]	2019	Statisk analys	5	Binärer
[269]	2018	Statisk analys	5	Android
[270]	2021	Fuzzer	5	Binärer

Tabell B.1. Granskade verktyg i fulltextanalys (forts.).

Referens	År	Typ av verktyg	TRL-nivå	Målmjukvara
[271]	2023	Fuzzer	5	Binärer
[272]	2017	Fuzzer	5	-
[273]	2021	Mönsterigenkänning	4	Binärer
[274]	2020	Fuzzer	6	-
[275]	2020	Fuzzer	5	Binärer
[276]	2015	Hybridmetoder	5	-
[277]	2018	Fuzzer	6	-
[278]	2021	Fuzzer	7	Javascript
[279]	2022	Fuzzer	5	-

ISSN 1650-1942

www.foi.se